Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Matej Zavrtanik

Primerjava sistemov za dodeljevanje pomnilnika v programskem jeziku C

diplomsko delo

visokošolski strokovni študijski program prve stopnje računalništvo in informatika

Mentor: doc. dr. Jurij Mihelič

Ljubljana, 2016

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo: Primerjava sistemov za dodeljevanje pomnilnika v programskem jeziku C

Tematika naloge:

Programski jezik C je zelo razširjen na področju razvoja sistemske programske opreme. Eden izmed razlogov za to je tudi ročno dodeljevanje pomnilnika, ki ga jezik podpira. Tekom časa se je pojavilo več različnih sistemov za dodeljevanje pomnilnika. V okviru diplomske naloge preglejte področje in opišite delovanje različnih sistemov za dodeljevanje pomnilnika. Izbrane sisteme eksperimentalno ovrednotite in primerjajte njihovo delovanje na različnih programih in testnih scenarijih.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Matej Zavrtanik sem avtor diplomskega dela z naslovom:

Primerjava sistemov za dodeljevanje pomnilnika v programskem jeziku C (angl. *Comparison of systems for memory allocation in the C programming language*)

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Jurija Miheliča,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 23. avgusta 2016 Podpis avtorja:

Zahvaljujem se vsem programerjem, ki so prispevali k nastanku odprtokodnih programov.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
ASLR	address space layout randomization	naključna razvrstitev v naslovnem prostoru
CPE		centralna procesna enota
FIFO	first-in first-out	prvi noter, prvi ven
LIFO	last-in first-out	zadnji noter, prvi ven
OS	operating system	operacijski sistem
POSIX	portable operating system interface	prenosljivi vmesnik za operacijske sisteme
TLB	translation lookaside buffer	medpomnilnik za prevajanje naslovov

Povzetek

Naslov: Primerjava sistemov za dodeljevanje pomnilnika v programskem jeziku C.

V diplomskem delu je opisano dodeljevanje pomnilnika. Na začetku dela so opisani uporabljeni mehanizmi, sistemski klici in podatkovne strukture v alokatorjih. Našteti so cilji dodeljevanja pomnilnika in težave, ki se jih morajo izogniti. V nadaljevanju so opisani zgradba in delovanje nekaterih alokatorjev, ki so v širši uporabi. Na koncu so alokatorji primerjani po času izvajanja in porabi pomnilnika. Na podlagi tega je izpeljan zaključek.

Ključne besede: računalnik, pomnilnik, dodeljevanje pomnilnika, programski jezik C, programiranje.

Abstract

Title: Comparison of systems for memory allocation in the C programming language.

The bachelor thesis describes memory allocation. Work begins with description of mechanism, system calls and data structures used in memory allocators. Goals of memory allocation ares listed along with problems which must be avoided. Afterwards construction and allocating of popular memory allocators is described. Work ends with comparison of memory allocators based on time of execution of programs and memory usage, on which conclusion is based.

Keywords: computer, memory, memory allocation, C programming language, programming.

Poglavje 1

Uvod

Z dodeljevanjem pomnilnika se danes programer sreča predvsem v programskem jeziku C, kjer za to uporabi funkcijo `malloc`. Dinamično dodeljevanje pomnilnika nalaga programerju dodatno delo, a mu tudi nudi večji nadzor nad izvajanjem programa in nad porabo pomnilnika. To je tudi en izmed razlogov, zakaj se programski jezik C še vedno široko uporablja, predvsem na področjih, kot so sistemska programska oprema, operacijski sistemi, sistemi, ki tečejo v realnem času, itd. Če se programer želi izogniti "ročnemu" dodeljevanju pomnilnika, lahko uporabi avtomatsko čiščenje pomnilnika (angl. garbage collector). Danes večina programskih jezikov uporablja avtomatsko čiščenje pomnilnika.

Področje raziskav dodeljevanja pomnilnika je zelo staro in sega v obdobje prvih računalnikov. Na začetku so se ukvarjali z iskanjem čim učinkovitejšega načina predstavitve blokov pomnilnika in njihovim iskanjem. Danes pa predstavlja izziv implementacija čim učinkovitejšega alokatorja na računalnikih, ki lahko poganjajo večje število niti hkrati.

Poglavje 2

Dodeljevanje pomnilnika

V tem poglavju je opisano dodeljevanje pomnilnika s teoretičnega vidika.

2.1 Pomnilnik

Delovanje pomnilnika na najnižji ravni ni pomembno za preučevanje dodeljevanja. Je pa pomembno, kako CPE uporablja pomnilnik.

2.1.1 Pomnilniška hierarhija

Na začetku so procesorji dostopali do pomnilnika neposredno, a je z leti pomnilnik postal prepočasen. Zato so v CPE dodali predpomnilnik, ki je bistveno hitrejši od glavnega pomnilnika, a tudi precej manjši. Predpomnilnik je sestavljen iz večjega števila nivojev [6]. Prvi nivo je najhitrejši in najmanjši, vsak dodatni nivo pa je nekoliko počasnejši in večji. Poleg razdelitve po nivojih je predpomnilnik razdeljen na ukazni in podatkovni del.

Ker pomnilnika lahko zmanjka, je na trdem disku na voljo še izmenjevalni prostor. Dostop do trdega diska je zelo počasen v primerjavi z dostopom do pomnilnika, in če so ti pogosti, je delovanje programa lahko nekajkrat počasnejše. V tabeli 2.1 so podani časi dostopa do predpomnilnikov in do glavnega pomnilnika. Podatki za čas so približni in veljajo za procesor Intel i7-4770 [3].

Mesto dostopa	**Čas v ciklih**	**Velikost**
Register	takoj	16 registrov po 8 B
Predpomnilnik L1	4-5	32 KiB na jedro
Predpomnilnik L2	12	256 KiB na jedro
Predpomnilnik L3	36	8 MiB za vsa jedra
Pomnilnik	230	Običajno med 4 GiB in 16 GiB

Tabela 2.1: Časi dostopov do pomnilnika.

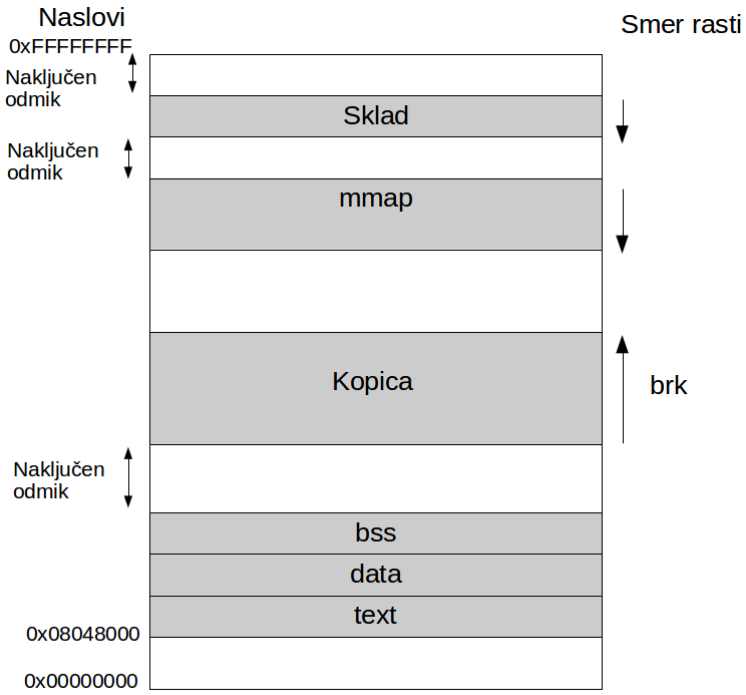
2.1.2 Strani

Naslavljanje glavnega pomnilnika se izvede posredno [21]. V programih se uporabljajo navidezni naslovi, ti pa se v enoti za upravljanje s pomnilnikom preslikajo v fizične naslove. Tako fizični pomnilnik kot navidezni pomnilnik sta razdeljena na strani enakih velikosti. Ker je navidezni naslovni prostor po navadi večji od fizičnega, lahko zmanjka prostih fizičnih strani pomnilnika. Takrat OS poišče strani, ki trenutno niso v uporabi, in jih zapiše na trdi disk. Ko pa je neka stran na izmenjevalnem prostoru na disku in jo program naslovi, mora OS ponovno poiskati nerabljeno stran, jo zapisati na disk, nato pa naslovljeno stran prenesti v pomnilnik.

Različne arhitekture dopuščajo različne velikosti strani. Ker večina alokatorjev od operacijskega sistema zahteva proste strani pomnilnika, je za prenosljivost alokatorja treba definirati različne velikosti strani. Večina arhitektur uporablja strani velikosti 4 KiB.

2.1.3 Segmentacija programa

Program v pomnilniku je razdeljen na večje število segmentov [7]. Segmenti programa so: tekst (*.text*), inicializirane spremenljivke (*.data*), neinicializirane spremenljivke (*.bss*), sklad, kopica in del, rezerviran z `mmap`, kamor spadajo tudi knjižnice. Ker so meje segmentov znane, lahko pri klicu funkcije `free` preverimo, ali je kazalec znotraj segmenta kopice ali segmentov,



Slika 2.1: Segmentacija programa.

pridobljenih z `mmap`. Segmentacija programa je razvidna iz slike 2.1.

Če skuša prebrati podatek zunaj kateregakoli segmenta ali pa skuša pisati podatek v segment, zavarovan za pisanje, pride do napake segmenta. Na sistemih UNIX operacijski sistem s signalom `SIGSEGV` ubije proces. Nekateri operacijski sistemi uporabljajo tehniko ASLR, ki naključno določi odmik med segmenti [1]. Odmik se določi pred izvajanjem programa. Namen tega je zagotoviti nekoliko boljšo varnost programov.

2.2 Sistemski klici za delo s pomnilnikom

Za pridobivanje pomnilnika se uporabljata sistemska klica `brk` in `mmap` [11] [8]. Razlike med njima so naslednje. Uporaba sistemskega klica `brk` je preprostejša in lahko zgolj povečuje segment kopice, medtem ko `mmap` z vsakim klicem ustvari nov segment. Sicer je mogoče dodati nov segment ob mejah

obstoječega, a je treba to podati kot argument funkciji. Nekoč se je uporabljal izključno za sistemski klice `brk`, danes pa se nekoliko več uporablja `mmap`.

Podpis ovojnih funkcij sistemskega klica `brk`:

```
int brk(void *addr);
void *sbrk(intptr_t increment);
```

Večina UNIX sistemov pozna še sistemski klic `mmap` in `munmap`. Podpisa obeh ovojnih funkcij:

```
void *mmap(void *addr, size_t length, int prot, int
    flags, int fd, off_t offset);
int munmap(void *addr, size_t length);
```

Povečanje in pomanjšanje kopice s sistemskim klicem `brk` bi izgledalo takole:

```
char *p;
p = sbrk(100); /* povečaj kopico za 100 B */
/* uporabi prostor na kopici */
sbrk(-100); /* zmanjšaj kopico za 100 B */
```

Povečanje kopice z uporabo `brk` se nekoliko spreminja od povečanja z uporabo `sbrk`. Večinoma se uporablja `brk`, ker alokator že pozna vrh kopice. V praksi je enota, s katero se povečuje kopico, enaka velikosti strani. Primera v kodi sta še vedno pravilna, a sta nekoliko neučinkovita.

```
char *heap, *p; /* heap je kazalec na vrh kopice. */
p = heap = sbrk(0); /* trenutni vrh kopice */
heap += 100;
brk(heap); /* povečaj kopico za 100 B */
/* uporabi prostor na kopici */
heap -= 100;
brk(heap); /* zmanjšaj kopico za 100 B */
```

Pridobivanje ene strani pomnilnika od sistema z uporabo `mmap` bi izgledalo takole:

```
char *p;
p = mmap(NULL, 4096, PROT_READ | PROT_WRITE,
    MAP_PRIVATE | MAP_ANONYMOUS, -1, 0); /* alociraj
    prostor */
/* uporabi prostor */
munmap(p, 4096); /* vrni prostor OS */
```

Operacijskemu sistemu je mogoče ukazati, kaj naj naredi z obsegom pomnilniških strani. Ukaz lahko vsebuje informacije o načinu dostopa, o pogostosti dostopa itd. Ukaz se poda s sistemskim klicem `madvise` iz zaglavja `sys/mman.h` [5]. Klic s parametrom *MADV_DONTNEED* povzroči, da bodo strani vrnjene sistemu. Vrnjene so na ta način, da je njihova vsebina prepisana s številom nič in ne vplivajo več na porabo. Ob naslednji referenci znotraj obsega pa ne pride do napake segmenta.

```
int madvise(void *addr, size_t length, int advice);
```

Takole pa bi za eno stran pomnilnika vrnili OS. Pozneje pa bi jo še vedno lahko uporabili.

```
char *p;
madvise(p, 4096, MADV_DONTNEED);
```

Poleg teh sistemskih klicev se uporabljajo za delo s pomnilnikom še drugi, kot sta na primer `mprotect`, ki se uporablja za nastavljanje pravic branja in pisanja na strani. To pride prav, ko za proste strani odvzamemo pravice branja in pisanja. Zato vsak dostop do teh strani povzroči signal `SIGSEGV`. Obstajajo tudi drugi sistemski klici za delo s pomnilnikom, a se v alokatorjih ne uporabljajo. Eden takih je tudi `mlock`, ki zaklene del pomnilnika in s tem prepreči, da bi ga OS dal na izmenjevalni prostor.

2.2.1 Časi izvajanja sistemskih klicev

V tabeli 2.2 so časi izvajanja za sistemske klice na sistemu Linux. V resnici je funkcija `sbrk` zgolj ovojna funkcija in dvakrat kliče sistemski klic `brk`.

Sistemski klic	**Čas izvajanja v ciklih**
brk	160
sbrk	394
mmap	602
munmap	993
madvise	364

Tabela 2.2: Časi izvajanja sistemskih klicev.

2.3 Funkcija malloc

V večini primerov dodeljevanje in sproščanje pomnilnika poteka z uporabo funkcij `malloc` in `free`. Najnovejši standard jezika C [12] definira naslednje funkcije za dodeljevanje in sproščanje pomnilnika:

```
void *aligned_alloc(size_t alignment, size_t size);
void *calloc(size_t nmemb, size_t size);
void free(void *ptr);
void *malloc(size_t size);
void *realloc(void *ptr, size_t size);
```

`malloc` Vrne kazalec na začetek dodeljenega prostora, v primeru napake pa vrne kazalec na `NULL`. Standard jezika C iz leta 2011 ne opisuje, kaj mora vrniti v primeru, da je podana zahteva za blok velikosti 0. Običajno alokatorji tako zahtevo obravnavajo kot najmanjši možni blok in ne vrnejo kazalca na `NULL`, ker se to razume kot napako.

`free` Vrne dodeljen blok alokatorju. Alokator lahko prost pomnilnik vrne operacijskemu sistemu ali pa ga obdrži za prihodnja dodeljevanja. Alokator blokov, manjših od velikosti strani, ne more v nobenem primeru vrniti sistemu, zato jih obdrži. V primeru, da podan kazalec ne kaže na začetek bloka, dodeljenega z eno izmed teh funkcij, obnašanje ni definirano. To je zelo pomemben detajl, saj omogoča hitrejše sproščanje

blokov, ker se alokator lahko izogne preverjanju ustreznosti naslova. Nekatere implementacije prepišejo vsebino prostih blokov z ničlami ali naključnimi števili. Razlog za to je nekoliko boljša varnost proti zlonamerni kodi. Standard tega ne zahteva in tak način delovanja je mogoče izklopiti.

`calloc` Vrne kazalec na dodeljen blok, prepisan z ničlami. V primeru napake vrne kazalec na `NULL`.

`realloc` Poveča ali pomanjša obstoječ blok pomnilnika. Če je le-ta premajhen, se dodeli nov blok ustrezne velikosti in prekopira staro vsebino na tisti naslov. S parametrom 0 deluje kot funkcija `free` ali pa vrne najmanjši možni blok tako kot `malloc`. Če je podan kazalec `NULL`, pa dodeli pomnilnik kot `malloc`. V primeru, da podan kazalec ne kaže na naslov enega izmed dodeljenih blokov, obnašanje ni definirano.

`aligned_alloc` Dodeli pomnilnik na ustrezno poravnanem naslovu. Običajno so naslovi poravnani na potenco števila dve.

Standard je zelo preprost in prepušča avtorjem alokatorja veliko svobode. Zato so si implementacije glede delovanja zelo različne.

Nekatere implementacije so tudi v skladu s standardom POSIX, ki definira še funkcijo `posix_memalign`. Funkcija deluje podobno kot `aligned_alloc`, le da ji moramo podati kazalec. Ob uspehu funkcija vrne število nič, sicer neničelno število za primer napake.

```
int posix_memalign(void **memptr, size_t alignment,
    size_t size);
```

2.3.1 Cilji funkcije malloc

Da je alokator primeren za splošno rabo, mora ustrezati čim večjemu številu ciljev. Nekateri izmed ciljev, ki jih navaja Doug Lea [18], so naslednji:

Prenosljivost Delovati mora na večjem številu operacijskih sistemov in na različnih arhitekturah.

Minimizira porabo pomnilnika Poraba pomnilnika mora biti sorazmerna dejanski porabi.

Hitrost Velikokrat odločilni faktor pri izbiri.

Lokalnost blokov Lokalnost podatkov pohitri delovanje programa.

Razširljivost Večanje števila niti nima prevelikega vpliva na hitrost delovanja ali porabo pomnilnika.

Omogoča nastavljanje parametrov S tem programerju omogoča in olajša delo pri optimiziranju funkcije.

Poveča možnost detekcije napak Omogoča lažje razhroščevanje programske kode in boljše razumevanje delovanja programa.

Predvidljivost delovanja Časi izvajanja klicev funkcij se ne smejo preveč razlikovati. Predvidljivost je pomembna za sisteme, ki delujejo v realnem času.

Splošnost Alokator se mora dobro obnesti za različne tipe programov.

2.4 Zgradba kopice

Bloki v pomnilniku so razvrščeni s pomočjo različnih podatkovnih struktur, skoraj vse pa na nekem mestu uporabljajo seznam. Vse implementacije uporabljajo različne kombinacije podatkovnih struktur, zato je delitev alokatorjev glede na uporabljene podatkovne strukture nesmiselna. Poleg seznama se uporabljajo še polja bitov, različne drevesne strukture, redko tudi zgoščevalna tabela in bločni sistem.

2.4.1 Povezan seznam

Povezani seznami so primerni za učinkovito predstavitev kopice. Vsako vozlišče v seznamu predstavlja en blok na kopici. Za vsak blok zadostujeta podatka o velikosti in o tem, ali je zaseden. Kazalec na naslednji blok ni potreben, saj je njegov naslov mogoče izračunati s pomočjo podatka o velikosti iz trenutnega bloka.

Prosti bloki so lahko razvrščeni na več načinov. Lahko so razvrščeni po velikosti, po času vračanja, naslovu itd. Pogosto so razvrščeni v večje število seznamov, znotraj posameznega seznama pa so vsi enake velikosti. V sezname se vnašajo v vrstnem redu FIFO ali LIFO glede na čas sproščanja blokov. Vnašanje v vrstnem redu FIFO zmanjšuje fragmentacijo [18]. Vnašanje v vrstnem redu LIFO pa je nekoliko hitrejše, saj je večja verjetnost, da bo ob brisanju s seznama blok že v predpomnilniku [13]. Razvrstitev po seznamih je lahko določena v programski kodi in se ne spreminja ali pa je spremenljiva z uporabo drevesnih podatkovnih struktur.

Prednost seznamov je ta, da so bloki skoraj poljubnih velikosti, za razliko od polja bitov, kjer so pogoste dovoljene velikosti potence števila dve. Zaradi tega imajo majhno notranjo fragmentacijo, a lahko pride do nekoliko večje zunanje fragmentacije. Slabost seznama v primerjavi s poljem bitov je ta, da mora za vsak blok hraniti glavo, medtem ko v bitni predstavitvi zadostuje zgolj en bit na blok.

Iskanje prostih blokov

Če so bloki znotraj seznama enakih velikosti, je že prvi blok ustrezen in iskanje nima smisla. Sicer je možno prost blok poiskati na več različnih načinov [21].

Prvo ujemanje Prvo ujemanje poišče prvi blok, ki je dovolj velik za zahtevo. V primeru, da je prazen blok zahtevane velikosti takoj na začetku, se bo iskanje zelo hitro zaključilo. Če uporabimo to iskanje na seznamu, urejenemu po naslovih, potem bodo dodeljeni bloki prevladovali na

nižjih naslovih in bo večja verjetnost, da bo prost pomnilnik na vrhu kopice vrnjen OS. Slabost tega iskanja je ta, da najde bloke neustrezne velikosti, zato se lahko zgodi, da je najdene bloke treba deliti na dva dela. Deljenje blokov povzroča fragmentacijo pomnilnika.

Naslednje ujemanje Poišče prvi dovolj velik prost blok od mesta, kjer se je prejšnje iskanje ustavilo. Naslednje ujemanje preprečuje fragmentacijo na začetku seznama, a se zaradi tega poveča skupna fragmentacija [15].

Najboljše ujemanje To iskanje poišče najmanjši in hkrati dovolj velik blok. Slabost iskanja na ta način je, da mora preiskati prav vse bloke, kar je lahko zelo potratno. Iskanje se lahko zaključi tudi prej, če najde blok točno zahtevane velikosti.

Najslabše ujemanje Kot rešitev za fragmentacijo so preizkusili tudi najslabše ujemanje. Predpostavka je bila, da najslabše ujemanje preprečuje fragmentacijo, ker vedno poišče največji prost blok. A iskanje na ta način ne odpravlja fragmentacije in je zelo počasno.

Združevanje in deljenje prostih blokov

Združevanje prostih blokov v seznamih je preprosto, če poznamo predhodnika ali naslednika. Združevanje blokov zmanjšuje fragmentacijo, a zahteva dodatno število operacij. Poleg tega pa bi lahko prišla takoj po vračanju zahteva po bloku enake velikosti in bi z združevanjem opravili nepotrebno delo. Pogosto je pri združevanju upoštevana tudi velikost blokov. Glavni cilj združevanja blokov je zmanjšanje fragmentacije kopice [22]. Združevanje prostih blokov se lahko zgodi ob naslednjih dogodkih:

Nikoli V tem primeru je možna večja fragmentacija, a je potrebnih nekoliko manj operacij pri sproščanju.

Ob vsakem sproščanju Za vsak nov prost blok se preveri, ali sta njegova soseda prosta; če sta, se združijo v en sam blok.

Na določeno število sproščanj Po preštetem številu sproščanj se za proste bloke preveri, ali jih je možno združiti.

Ko kopice ni mogoče povečati Ko ni mogoče najti prostega bloka in ko od OS ni mogoče zahtevati prostega pomnilnika, se alokator loti združevanja prostih blokov.

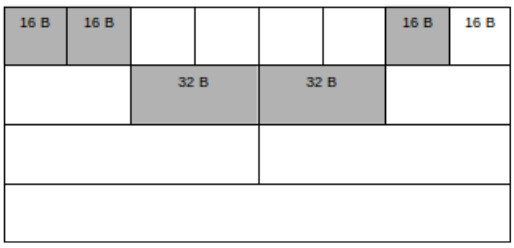
Z deljenjem blokov se zmanjšuje notranjo fragmentacijo. A tudi deljenje blokov povsem ne prepreči fragmentacije, saj se lahko zgodi, da so ostanki premajhni, da bi koristili zahtevam. Zaradi boljše lokalnosti nekateri alokatorji ohranijo ostanek pri deljenju zadnjega bloka. Če je ta dovolj velik, se ga uporabi pri naslednjem deljenju.

Optimizacije seznama

Zelo pogosta optimizacija je ta, da mora biti za vse bloke velikost zaokrožena na število, ki je deljivo s štiri ali osem (lahko tudi več, pogoj je, da je število potenca števila dve). S tem zagotovimo, da sta zadnja dva ali trije biti vedno enaki nič [18]. Zato te bite lahko učinkovito uporabimo za označevanje, ali je blok zaseden. Zato je treba pri branju števila maskirati zadnje bite, da dobimo pravo velikost bloka.

Optimizacijo, imenovano mejne značke (angl. boundary tags), je predlagal Donald Knuth [17]. Poleg bita o zasedenosti trenutnega bloka imamo na najnižjih bitih glave bloka še bit, ki označuje zasedenost prejšnjega bloka. V primeru, da je prejšnji blok prost, je velikost prejšnjega bloka podana pred glavo trenutnega bloka. S tem je omogočeno združevanje trenutnega bloka z morebitnim prostim predhodnikom v konstantnem času in brez dodatne porabe pomnilnika. Brez mejnih značk bi bilo mogoče edino združevanje z nasledniki. Primer mejne značke je mogoče videti na sliki 3.2.

Če je velikost najmanjšega bloka dovolj velika, da lahko hrani dva kazalca in število [18], potem lahko prazen prostor znotraj prostih blokov poleg mejnih značk uporabimo še za kazalca na prejšnji in naslednji prosti blok v seznamu.



Slika 2.2: Zgradba bločnega sistema velikosti 128 B.

2.4.2 Polje bitov

V polju bitov so zasedeni bloki predstavljeni s številom ena, prosti pa z ničlo oziroma obratno [21]. Vsi bloki v bitni predstavitvi so enake velikosti, pogosto je to potenca števila dve. Iskanje prostega bloka zahteva iskanje prve ničle v bitnem polju. Blok se dodeli s spremembo bita na ena, sprosti pa s spremembo na nič. Če je zahteva večja od velikosti bloka, je treba poiskati dovolj dolgo zaporedje ničel v bitnem polju, zato se v nekaterih primerih proste bloke, če so dovolj veliki, vstavi na seznam prostih blokov. Za večje bloke se polje bitov redkeje uporablja. Polje bitov ima to prednost, da se za vsak blok porabi samo en dodaten bit, a se lahko po drugi strani zgodi, da se rezervira celotno polje zgolj za majhno število blokov.

2.4.3 Bločni sistem

Obstaja več različic bločnega sistema, najpogostejša pa je različica dvojiškega bločnega sistema, kjer so vsi bloki velikosti potence števila dve. Ob prvi dodelitvi se blok deli na polovico toliko časa, dokler ja ta dovolj velik za zahtevo. Npr. če je velikost dvojiškega bločnega sistema na začetku 1024 B in rezerviramo prostor za 200 B, bomo najprej 1024 B razdelili na pol, nato pa 512 B razdelili še enkrat na pol, da dobimo 256 B velik blok. Po teh delitvah nam ostaneta 512 B in 256 B velika prosta bloka. Na sliki 2.2 je primer bločnega sistema velikosti 128 B, z enim prostim blokom.

Po vračanju dodeljenega bloka se prosti bloki med sabo združujejo. A združijo se lahko le z istim blokom, kot so se prej delili na pol. Prosti bloki

so povezani v sezname, znotraj seznama pa so vsi enake velikosti. Prostor znotraj bloka pa se da uporabiti za kazalca na prejšnji in naslednji prost blok iste velikosti [2].

Glavna slabost dvojiški bločnega sistema je velika notranja fragmentacija, ta je v povprečju 25%. To so skušali odpraviti z implementacijo dvojnega bločnega sistema, kjer so bloki velikosti 2, 4, 8, 6 itd. in velikosti 3, 6, 12, 24 itd. S tem se poveča število možnih velikosti za 2, kar zmanjša notranjo fragmentacijo [22].

Druga varianta je Fibonaccijev bločni sistem, kjer so velikosti blokov števila iz Fibonaccijevega zaporedja [22]. Ker je vsako Fibonaccijevo število vsota dveh manjših Fibonaccijevih števil, se da bloke enostavno deliti. Na primer, predpostavimo, da je začetni blok velikosti 34, kar je število iz Fibonaccijevega zaporedja, želimo pa blok velikosti 6. Najprej 34 delimo na 21 in 13. Nato 13 delimo na 8 in 5, kjer je blok velikosti 8 dovolj velik za podano zahtevo.

2.5 Poraba pomnilnika

Poraba pomnilnika pri programih ima neko obliko. Teh oblik ni veliko, najpogostejše pa so tukaj naštete. Oblika porabe se tukaj nanaša na porabo, ki jo program zahteva od alokatorja, in ne na skupno porabo, vključno s fragmentacijo [22].

Vrh Količina zahtevanega pomnilnika se s časom povečuje. Fragmentacija pomnilnika je pomembna, a običajno alokatorji nimajo težav s fragmentacijo, če zgolj dodeljujejo pomnilnik in ga ne sproščajo.

Planota Program pridobi ves pomnilnik na začetku izvajanja, nato se poraba ne spreminja več. Pri teh programih je hitrost dodeljevanja skoraj zanemarljiva. Pomembnejša je lokalnost podatkov.

Vrhovi Program gre skozi različne faze, kjer se na začetku faze dodeli pomnilnik, na koncu faze pa sprosti. Pri teh programih so pomembni

hitrost, lokalnost in fragmentacija. Takšni programi tudi največ pridobijo z izbiro boljšega alokatorja. Če so faze zelo kratke, vračanje pomnilnika ni tako zelo pomembno, saj nekajsekundna sprostitev pomnilnika ne koristi nikomur.

Možne so tudi druge oblike oziroma kombinacije med naštetimi. Npr. program ima na začetku obliko vrhov, nato se poraba ustali in dobi obliko planote [22].

2.5.1 Vračanje pomnilnika OS

Pogosto je vračanje pomnilnika OS nemogoče, ker to onemogoča zaseden blok na vrhu kopice. A ker so sistemski klici počasni, alokator ne vrača pomnilnika, dokler velikost tega prostora ne presega določene meje. Zato je določitev meje, po kateri bo alokator začel vračati pomnilnik, pomembna, saj vpliva na porabo pomnilnika in na hitrost programa.

2.6 Strategije

Do zdaj smo opisali nekatere mehanizme in pristope. Da pri implementaciji izberemo prave mehanizme in pristope, je smiselno definirati strategijo.

- Bloki, dodeljeni v istem časovnem obdobju, bodo tudi sproščeni v istem časovnem obdobju. [15]
- Bloki enakih velikosti so verjetno v uporabi iste podatkovne strukture, zato bodo uporabljeni v istih funkcijah. Bloke enakih velikosti je zato smiselno postaviti skupaj zaradi boljše lokalnosti.
- Ko program zahteva blok določene velikosti, bo tej zahtevi verjetneje sledila zahteva po bloku enake velikosti.
- Programi uporabljajo majhno število različnih velikosti blokov. [15]
- Ista nit, ki bo pridobila bloke, jih bo tudi uporabljala.

2.7 Alokatorji po meri

Nekateri programi uporabljajo tudi alokatorje po meri. Glavna razloga za to sta hitrejše izvajanje programa ali manjša poraba pomnilnika. V nekaterih primerih takšni alokatorji pomembno vplivajo na hitrost izvajanja in porabo pomnilnika. V večini primerov se ti alokatorji ne uporabljajo, razen če se ne posebej izkaže, da je dodeljevanje pomnilnika ozko grlo pri izvajanju [9].

2.7.1 Razredni alokator

Deluje podobno kot običajen splošno namenski alokator, a ker je večina blokov enake velikosti, le-te obravnava posebej od preostalih. Ti bloki so običajno predstavljeni s poljem bitov, da je izguba pomnilnika čim manjša. Razredni alokator je mogoče implementirati v ovojni funkciji nad `malloc` funkcijo. Razredni alokator je smiselno implementirati v primeru, da splošni alokator za velikosti blokov uporablja potence števila dve, medtem ko zahteve povzročijo veliko notranjo fragmentacijo. Na primer program večinoma zahteva bloke velikosti 160 B, alokator pa bi uporabil bloke velikosti 256 B. V tem primeru bi notranja fragmentacija znašala kar 38%. Namesto tega bi lahko dodelili večje polje, kjer bi bili bloki velikosti 160 B.

2.7.2 Regijski alokator

Regijski alokator rezervira večji blok pomnilnika, nato pa se z dodeljevanji povečuje kazalec, ki kaže na prvi prost naslov. Ob dodeljevanju se preveri, ali je v regiji še dovolj prostora. Če ga ni, se dodeli nov blok pomnilnika, ki se nato uporablja za regijski alokator. Posameznih blokov ni mogoče označiti kot proste. To je mogoče storiti samo za celotno regijo. Regijski alokator ima dve prednosti, da dodeljeni bloki nimajo presežka v obliki podatka o velikosti bloka in da je dodeljevanje hitro z dobro lokalnostjo blokov. Ta alokator je smiselno uporabiti za bloke, dodeljene za krajše časovno obdobje. Ker ima regija fiksno določeno velikost, mora imeti kazalec na naslednjo regijo

v primeru, da v trenutni zmanjka prostega prostora. Regije se uporabljajo tudi v nekaterih splošno namenskih alokatorjih.

```
struct region {
  void *free_ptr; // kazalec na prosti naslov
  void *end_region; // kazalec na konec regije
  struct region *next; // kazalec na naslednjo regijo
};
```

Poleg navadnega regijskega alokatorja so možne še nekoliko spremenjene variante.

Gnezdene regije So nekoliko spremenjen regijski alokator. Poleg dodeljenih blokov imajo regije lahko še svoje podregije. To poenostavi upravljanje z večjim številom regij. Če se sprosti korensko regijo, se sprosti ves pomnilnik v vseh podregijah in tudi znotraj regije same.

Regije s štetjem referenc V tej različici regijskega alokatorja ni treba posebej označiti proste regije, ampak se to določi s štetjem dodeljevanj in sproščanj. Če je razlika med številom klicev za dodeljevanje in sproščanje enaka 0, je regija označena kot prosta.

Sklad objektov [1] Je spremenjen regijski alokator, ki dopušča vračanje prostora v primeru, da si dodeljevanje in sproščanje blokov sledita v vrstnem redu LIFO. Pri sproščanju se izvede zgolj sprememba kazalca, ki kaže na prvi prost naslov, na nižji naslov. S tem je območje med novim in starim naslovom sproščeno.

2.7.3 Dodeljevanje na skladu

V programskem jeziku C lahko s funkcijo `alloca` dodelimo prostor na skladu. Funkcija `alloca` je v primerjavi s funkcijo `malloc` hitrejša, a ima dodeljevanje na skladu veliko slabosti: sklad je v primerjavi s kopico bistveno bolj

[1] angl. obstack

omejen, prekoračitev sklada pa povzroči sesutje programa. Sproščanje pomnilnika se izvede samo od sebe po izhodu iz funkcije, kar je tudi ena izmed prednosti dodeljevanja na skladu. A slabosti funkcije `alloca` odtehtajo njene prednosti, zato se jo poredko uporablja.

2.8 Dodeljevanje v večnitnih aplikacijah

Reševanje problema dodeljevanja pomnilnika za večnitne programe ima najverjetneje največji vpliv na načrtovanje in izvedbo alokatorja. Starejši alokatorji so rešili ta problem enostavno, a neučinkovito. Problem so rešili z uporabo ene ključavnice, ki je omogočala le eno dodeljevanje ali sproščanje naenkrat. Ta problem se da učinkoviteje rešiti z uporabo aren, predpomnilnika za nit ali s sočasnimi podatkovnimi strukturami. Pogosta je tudi kombinacija aren in predpomnilnika za nit.

2.8.1 Alokatorji z arenami

Namesto da za vsako dodeljevanje ali sproščanje pomnilnika zaklenemo ključavnico nad celotno kopico, to storimo samo nad določenim predelom, imenovanim arena. Vsaka arena upravlja s svojimi prostimi bloki in za vsak prost blok velja, da se nahaja v eni areni.

Ker areni lahko zmanjka prostora, ga mora ali pridobiti od sistema ali pa od drugih aren. Zato si aren ne smemo predstavljati kot enega velikega dela pomnilnika, znotraj katerega so prosti in zasedeni bloki, ampak kot sestavljeno iz večjega števila delov, razpršenih po pomnilniku. Ti deli pomnilnika imajo različna poimenovanja, kot so npr. podkopica ali superblok. Arene so lahko tudi fiksne velikosti in se v primeru polne zasedenosti ustvari novo areno, kar je nekoliko bolj nenavadna in manj učinkovita rešitev od aren, ki se lahko širijo.

Do arene lahko dostopa več niti, naenkrat pa lahko zgolj ena. Zato mora za vsako dodeljevanje in sproščanje blokov zakleniti ključavnico nad areno. Primerno število aren je najmanj toliko, kolikor niti lahko procesor naenkrat

izvaja, oziroma neki večkratnik tega števila. S tem se zmanjša verjetnost, da bi bile vse arene zasedene, ker so niti prešle v spanje med dodeljevanjem ali sproščanjem pomnilnika.

Izbiranje aren

Nit lahko izbere ustrezno areno na več načinov. Zaradi boljše lokalnosti podatkov si je smiselno za vsako nit zapomniti areno. Če je ta zasedena, pa je treba poiskati prosto ali ustvariti novo, če je število aren še dovolj nizko. Če alokator skuša poiskati novo areno, lahko izbere prvo prosto, ki je trenutno na voljo, ali pa s pomočjo statistike uporabe aren tako, ki je najmanj v uporabi in je trenutno prosta. Če pa so vse arene zasedene, potem mora dodeljevanje za to nit počakati, da se bo sprostila ena izmed zasedenih aren.

2.8.2 Alokatorji s predpomnilnikom za nit

Predpomnilnik za nit je na prvi pogled podoben arenam, a je med njima veliko pomembnih razlik. Predpomnilnik za nit je zgrajen približno takole: za vsako nit hranimo več seznamov, na vsakem pa so vsi bloki iste velikosti. Ko pride do zahteve po novem bloku, se najprej preveri, ali je morda že kakšen prost blok na katerem izmed seznamov. Če ga ni, je treba pridobiti dodaten pomnilnik od sistema ali pa imeti glavno areno, ki hrani višek prostih blokov in jih po potrebi posreduje predpomnilnikom. Vsaka nit lahko dostopa le do svojega predpomnilnika, za kar pa ne rabi ključavnic. Za dostopanje do glavne arene pa rabi ključavnico. Glavna arena se uporablja za to, da niti vračajo presežek prostih blokov, in za to, da se ob končanju izvajanja niti lahko zbere proste bloke za ponovno uporabo.

En izmed problemov, ki nastane pri uporabi predpomnilnika, je ta, kaj storiti, ko ena nit zahteva nove bloke, neka druga pa jih vrača. Če rešitev ne uporablja glavnih aren, se morajo taki prosti bloki vrniti v predpomnilnik niti, ki je zahtevala ta blok, sicer lahko pride do neomejene porabe pomnilnika. Drug način je ta, da take proste bloke preprosto vrnemo v glavno

areno. V obeh primerih mora obstajati mehanizem, s katerim določimo lastništvo nad blokom. Ker v prvem primeru nit vrača blok v tuj predpomnilnik, nastane tu nova težava, saj druge niti ne smejo dostopati do tujega predpomnilnika.

Težavo se elegantno reši z uporabo sočasne vrste. Vsak predpomnilnik ima svojo sočasno vrsto, kamor preostale niti vračajo proste bloke, ki jih je zasedla ta nit [4]. Ko v predpomnilniku zmanjka prostih blokov, se preveri, ali so morda v vrsti na voljo prosti bloki. S tem se izognemo ključavnicam in dostopom do glavne arene, kar je tudi cilj predpomnilnika za nit.

Predpomnilnik za niti velja za zelo hitro rešitev, saj se skoraj v celoti izognemo ključavnicam in podatki imajo zelo dobro lokalnost. Ključna razlika med predpomnilnikom in arenami je ta, da se izognemo ključavnicam in s tem pohitrimo delovanje. A ker je število niti lahko zelo veliko, lahko v primeru uporabe predpomnilnika pride do nekoliko večje fragmentacije kopice. V tabeli 2.3 so podane ključne razlike med arenami in predpomnilnikom.

Lastnost	**Arene**	**Predpomnilnik**
Število	Toliko kot ima CPE jeder.	Toliko kot je število niti, ki se izvajajo.
Brisanje	Lokalno areno se lahko pobriše, če nima blokov v uporabi.	Ko nit zaključi izvajanje se pobriše njen predpomnilnik.
Dostop	Iz ene arene lahko pridobivajo bloke različne niti. Nit lahko zamenja areno.	Nit lahko dostopa samo do svojega predpomnilnika, ki ga ne more zamenjati za drugega.
Ključavnice	Za dostop se uporabljajo ključavnice.	Za dostop ključavnice niso potrebne.

Tabela 2.3: Primerjava aren in predpomnilnika.

2.8.3 Alokatorji s sočasnimi podatkovnimi strukturami

Alokatorji, ki zanašajo sočasne podatkovne strukture in so brez aren ali predpomnilnika za nit, so zelo redki. Obstaja sicer več opisov v literaturi, a samih implementacij je zelo malo (najbrž zato, ker so patentirani). Glavni sočasni podatkovni strukturi, ki se uporabljata, sta sočasna vrsta in sočasno B-drevo [13].

2.9 Problemi dodeljevanja

Pri dodeljevanju pomnilnika lahko pride do različnih problemov, ki se jim je treba izogniti. Fragmentaciji se sicer ni mogoče izogniti, se je pa zato mogoče izogniti problemom, ki nastanejo iz dodeljevanj tipa proizvajalec – porabniki.

2.9.1 Fragmentacija

Neuporabljen prostor znotraj blokov in med zasedenimi bloki imenujemo fragmentacija. Notranja fragmentacija je prostor, ki ostane neizkoriščen znotraj nekega bloka. Zunanja fragmentacija je ves neizkoriščen prostor zunaj zasedenih blokov. To vključuje tudi vse podatkovne strukture, ki jih uporablja alokator. Za reševanje notranje fragmentacije se uporablja deljenje prostih blokov. Težava, ki jo povzroči deljenje, je ta, da lahko nastane veliko število majhnih blokov, ki ne bodo nikoli dodeljeni. Proti zunanji fragmentaciji se uporablja združevanje sosednjih prostih blokov. Čeprav je to lahko učinkovita metoda, pa lahko privede do nepotrebnega združevanja blokov, saj bi ta blok lahko ustrezal prihodnji zahtevi.

Fragmentacijo povzroči več dejavnikov [22]. Zelo pogost vzrok fragmentacije so prosti bloki med dvema zasedenima blokoma. V tem primeru združevanje prostih blokov ni mogoče. Če bodo prihodnje zahteve po pomnilniku manjše od tega bloka, bo prišlo do notranje fragmentacije, če bodo večje, pa bo prišlo do zunanje fragmentacije. Do fragmentacije pride pogosto v programih, ki se izvajajo po fazah. V neki fazi prevladujejo dodeljevanja manjših

blokov, v neki drugi fazi pa večjih, kar lahko povzroči fragmentacijo. Fragmentacija je najbolj kritična ob najvišji porabi pomnilnika. Zmanjševanje fragmentacije je pomembno tudi zato, ker izboljša lokalnost podatkov [15].

2.9.2 Napačna skupna raba

Do napačne skupne rabe pride, ko različne niti berejo in spreminjajo različne podatke znotraj iste vrstice predpomnilnika[2]. Ko ena izmed niti spremeni podatek, se morajo podatki spremeniti tudi v predpomnilniku za drugo nit, čeprav le-ta do njih ne dostopa. Ta nepotrebna sinhronizacija predpomnilnika vpliva na hitrost delovanja programa. Napačno skupno rabo povzročajo alokatorji, ki sosednje bloke velikosti do 32 B dodelijo različnim nitim.

2.9.3 Zaklepanje

Alokatorji, ki uporabljajo ključavnice, imajo zaradi njih več morebitnih težav [19]. Ni nujno, da se opisane težave nanašajo na alokator, ki uporablja ključavnice.

Čakanje Če je ključavnica nad kopico zaklenjena, mora nit, ki skuša pridobiti pomnilnik, čakati. Čakanju se nit izogne tako, da je kopica razdeljena na več aren ali pa se uporabi alokator brez ključavnic.

Asinhroni signali V primeru, da nit skuša pridobiti pomnilnik, hkrati pa pride signal, ki aktivira funkcijo, ki ujame signal (angl. signal handler). Kopica je zaklenjena, zato ta funkcija ne sme pridobiti pomnilnika, sicer lahko pride do stradanja.

Nit, ubita med dodeljevanjem ali sproščanjem Če je nit ubita med dodeljevanjem ali sproščanjem, ima to lahko več posledic. To da nekateri bloki ne bodo sproščeni, velja tudi za alokatorje brez ključavnic. Zgodi

[2]Vrstice predpomnilnika so tudi znane kot bloki oziroma v angl. cache line. Zaradi manjše zmede uporabljamo izraz vrstica.

se lahko tudi to, da ostane kopica ali arena zaklenjena, kar ustavi vse niti, ki bi skušale pridobiti ali vrniti blok.

2.9.4 Problem dodeljevanj tipa proizvajalec – porabniki

V članku [13] je opisan problem, ki lahko nastane ob uporabi nekaterih implementacij v večnitnih programih. Do tega problema pride v alokatorjih z arenami. Do problema pride v primeru, ko ena nit pridobiva proste bloke, druge niti pa jih sproščajo. Zgradba alokatorja je takšna, da se prosti bloki ne vrnejo k niti, ki jih je zahtevala, ampak ostanejo v arenah niti, ki so jih sprostile. Tako nit, ki zaseda nove bloke, ne more uporabiti prostih blokov iz drugih aren in mora od sistema zahtevati več pomnilnika. Poraba s tem neomejeno narašča, kljub temu da bi zadostovala omejena količina pomnilnika.

2.10 Razhroščevanje in nastavljivost

Alokatorji poleg dodeljevanja in sproščanja lahko pomagajo pri razhroščevanju, profiliranju in merjenju porabe. Ker funkcije, ki niso namenjene dodeljevanju, niso standarizirane, se te funkcionalnosti zelo razlikujejo med alokatorji. Veliko alokatorjev podpira način razhroščevanja, ki opozori za primere večkratnega vračanja istega bloka, ali pa je pri sproščanju podan kazalec, ki ne kaže na dodeljen blok. Omogočajo tudi lažje zaznavanje prekoračitve pomnilnika z dodajanjem posebnega polja na koncu vsakega bloka.

Alokatorji lahko merijo porabo pomnilnika in število dodeljenih in prostih blokov. Te statistične podatke je možno od alokatorja pridobiti s klicem funkcije `mallinfo`, ki ni del standarda jezika C ali POSIX. Kljub temu je vsebina za strukturo povsod enaka.

```
struct mallinfo {
  int arena; // količina prostora brez mmap (v bajtih)
  int ordblks; // število prostih blokov
```

```
    int smblks; // količina prostih blokov v fastbin
    int hblks; // število blokov dodeljenih z mmap
    int hblkhd; // količina dodeljenega prostora z mmap (
        v bajtih)
    int usmblks; // največja količina dodeljenega
        prostora (v bajtih)
    int fsmblks; // količina sproščenega prostora v
        fastbin (v bajtih)
    int uordblks; // količina dodeljenega prostora (v
        bajtih)
    int fordblks; // količina prostega prostora (v bajtih
        )
    int keepcost; // količina prostega pomnilnika na vrhu
         kopice (v bajtih)
};
```

Veliko alokatorjev omogoča nastavljanje parametrov, ki vplivajo na delovanje. Žal za to ni standariziranega vmesnika. Večinoma se to opravi s klicem funkcije ali pa ob inicializaciji alokatorja z branjem iz datoteke `/etc/malloc.conf`. S parametri je mogoče vklopiti način razhroščevanja ali zbiranja statistik uporabe ali pa nastavljati prag vračanja prostih blokov sistemu.

Poglavje 3

Implementacije funkcije malloc

V tem poglavju so na kratko opisani nekateri izmed bolj znanih in uporabljenih alokatorjev. V tabeli 3.1 so zbrani tudi alokatorji, ki niso opisani, a se kljub temu uporabljajo.

Ime	**Jezik**	**Licenca**	**Uporaba**
dlmalloc	C	CC 1.0	Osnova za druge implementacije
phkmalloc	C	BEER-WARE	Nekoč del sistemov FreeBSD, OpenBSD
jemalloc	C	BSD-2	FreeBSD, Firefox
tcmalloc	C++	BSD-3	Google perftools
GNUmalloc	C	GPL	Del knjižnice GNU, odprtokodni projekti
nedmalloc	C	MIT	?
ottomalloc	C	MIT	OpenBSD
Hoard	C++	GNU GPLv2	?
Lockless	C	GNU GPLv3	?

Tabela 3.1: Implementacije malloca.

Veliko alokatorjev ima na začetku imena kratice svojih avtorjev. Na pri-

mer avtor dlmalloca je Doug Lea in avtor jemalloca je Jason Evans. Kar pa ne velja za tcmalloc, kjer kratica stoji za thread-cache.

3.1 phkmalloc

Poul-Henning Kamp je napisal phkmalloc v 90-ih letih za sistem FreeBSD. Ta alokator je ostal del sistema FreeBSD do leta 2006, ko ga je zaradi boljšega delovanja na večprocesorskih sistemih zamenjal jemalloc.

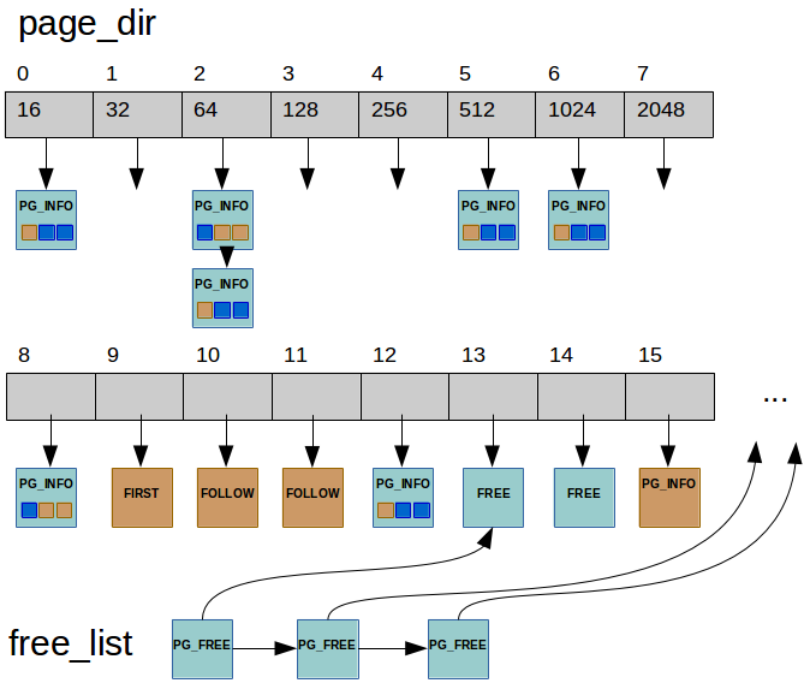
3.1.1 Zgradba kopice

Phkmalloc ima kopico razdeljeno na strani, podatki za manjše bloke pa se vodijo za vsako stran posebej. Zato bi lahko rekli, da skuša upravljati s stranmi in ne z bloki [16]. Motivacija za takšno izvedbo je bila ta, da je v uporabi čim manjše število strani in da pri iskanju obišče čim manjše število strani. Vsaki strani pripada kazalec na strukturo *pginfo* v polju *page_dir*. Polje *page_dir* je zasedeno z `mmap`. Za vse uporabnikove zahteve se ne glede na njihovo velikost uporablja zgolj sistemski klic `brk`.

Proste strani nimajo kazalca na strukturo *pginfo* v polju *page_dir*, ampak imajo kazalec na *MALLOC_FREE*. Bloki, ki zasedejo več kot polovico strani, imajo namesto kazalca na strukturo *pginfo* kazalec na *MALLOC_FIRST*, nato pa vse morebitne naslednje strani (če zasede več kot eno stran) imajo kazalec na *MALLOC_FOLLOW*. Le strani, ki vsebujejo bloke, manjše od polovice strani, imajo strukturo *pginfo*.

Znotraj strani so vsi bloki enake velikosti, njihova zasedenost pa se beleži v polju *bits* znotraj strukture *pginfo*. Velikost blokov, manjših od polovice strani, je vedno potenca števila dve. Najmanjši blok je privzeto velikosti 16 B, vsak naslednji je dvakrat večji od prejšnjega do velikosti polovice strani, kar je v večini primerov 2048 B. Zato je notranja fragmentacija velika, v povprečju 25%, a se ta kompenzira z manjšo porabo pomnilnika za označevanje manjših blokov zaradi polja bitov in z manjšo zunanjo fragmentacijo.

Strani z manjšimi bloki so vstavljene na seznam. Teh seznamov je toliko,



Slika 3.1: Zgradba phkmalloca.

kolikor je različnih velikosti blokov [16]. Teh seznamov je osem. Za dostop do teh seznamov se uporablja začetek polja *page_dir*, preostanek pa služi za dostop do struktur *pginfo*. Zato je za dostop do posamezne strani treba prišteti število osem. To je tudi razvidno s slike 3.1.

Za neki naslov se takole izračuna indeks v polju *page_dir*, če je a podan naslov in ps velikost strani pomnilnika, potem se ustrezen kazalec nahaja na indeksu i, ki ga izračunamo takole: $i = a/ps + 8$

Strukturi

Strukturo *pginfo* uporabljajo zasedene strani z manjšimi bloki. Struktura ni znotraj iste strani, kot so bloki, ampak je prostor zanjo dodeljen, tako kot za vse druge bloke. Nekoliko nenavadno, a phkmalloc kliče funkcijo `malloc` znotraj funkcije `malloc`.

```
struct pginfo {
```

```
  struct pginfo *next; // kazalec na naslednjo stran s
     prostimi bloki
  void *page; // kazalec na naslednjo stran
  u_short size; // velikost blokov na strani
  u_short shift; // velikost zamika za te bloke
  u_short free; // število prostih blokov
  u_short total; // število vseh blokov
  u_int bits[1]; // polje bitov prostih blokov
};
```

S strukturo *pgfree* so predstavljeni obsegi praznih strani.

```
struct pgfree {
  struct pgfree *next; // naslednji obseg prostih
     strani
  struct pgfree *prev; // prejšnji obseg prostih strani
  void *page; // kazalec na začetek prostih strani
  void *end; // kazalec na konec prostih strani
  size_t size; // število prostih bajtov
};
```

3.1.2 Dodeljevanje in sproščanje blokov

Ob dodeljevanju se preveri, ali je kakšna stran s prostimi bloki za dano velikost. Če je ni, alokator poišče prosto stran ali pa od OS zahteva za eno stran pomnilnika. Temu sledi iskanje prostega bloka v polju bitov. Iskanje je zagotovo uspešno, saj so strani brez prostih blokov odstranjene s seznama strani s prostimi bloki.

Ob klicu funkcije `free` se iz kazalca po zgornji formuli izračuna indeks v polju *page_dir*. Če je kazalec na *MALLOC_FIRST*, ga spremeni na kazalec *MALLOC_FREE*, isto naredi za vse naslednje strani, če imajo kazalec na *MALLOC_FOLLOW*. Za celoten obseg strani se ustvari struktura *pgfree*, ki je dodana na seznam prostih strani. Za manjše bloke kaže kazalec na strukturo

pginfo, kjer se ponastavi ustrezen bit. Če je bil sproščen zadnji blok znotraj strani, pomeni, da je stran prosta, zato je v polju *page_dir* označena s *MALLOC_FREE* in dodana na seznam prostih strani. V vseh drugih primerih pa je podani kazalec napačen.

Proste strani so hranjene v seznamu *free_list*, ki je urejen po naslovih naraščajoče. Za večje število zaporednih prostih strani zadostuje ena struktura *pgfree*. Za iskanje se uporablja prvo ujemanje zaradi hitrosti in ker ostajajo prosti bloki pri vrhu kopice, kar poveča možnost vračanja prostora OS. Ena izmed možnih nastavitev tega alokatorja je ta, da proste strani vrne s sistemskim klicem `madvise`.

Ker so strani urejene po naslovih, ima vstavljanje strukture *pgfree* na seznam *free_list* časovno zahtevnost $O(n)$, kjer je n v najslabšem primeru sorazmeren številu prostih strani. A ker alokator išče strani od nižjih naslovov in zaporedne proste strani označene z eno strukturo, je vstavljanje nekoliko hitrejše. Tudi v tem primeru iskanja dovolj dolgega obsega je časovna zahtevnost $O(n)$ in v primeru neuspeha preišče celoten seznam. Iz tega sledi, da so dodeljevanja za bloke, večje od ene strani, počasna.

3.1.3 Zaključek

phkmalloc je bil do prihoda večprocesorskih računalnikov en najboljših alokatorjev. phkmalloc uporablja na prvi pogled nekoliko neučinkovit sistem upravljanja z bloki, a se je v praksi pokazalo, da ni imel težav s fragmentacijo. V primerjavi s preostalimi alokatorji ima majhno število vrstic kode, ki je povrh dokaj preprosta. Naslednik je ottomalloc [20], ki kot osnovo uporablja kodo phkmalloca.

3.2 GNUmalloc

GNUmalloc je nekoliko spremenjen ptmalloc2, ta pa temelji na dlmallocu. Glavna razlika med GNUmallocom in dlmallocom je ta, da GNUmalloc uporablja arene za sočasno dodeljevanje. Ker je GNUmalloc del knjižnice GNU,

je najbrž en najbolj razširjenih alokatorjev. Znan je tudi pod drugimi imeni, kot je npr. glibcmalloc.

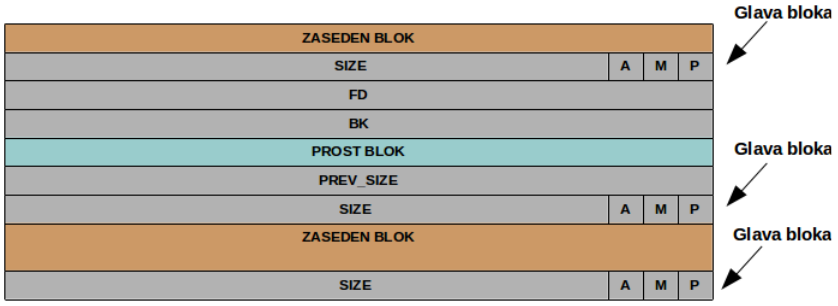
3.2.1 Bloki

GNUmalloc ima zasedene in proste bloke razvrščene po seznamih. Polj bitov ali bločnih sistemov se ne uporablja.

Zgradba bloka

Za vse bloke se uporablja strukturo *malloc_chunk*. V resnici se za zasedene bloke uporablja zgolj polje *size*, preostala polja pa so v uporabi v prostih blokih. Ker je za vsak blok velikost zaokrožena na število, deljivo z osem, je mogoče uporabiti zadnje tri bite polja *size*. Najmanj pomemben bit se uporablja za označitev, ali je prejšnji blok zaseden. Če ni zaseden, lahko iz polja *prev_size* preberemo njegovo velikost. Sicer do polja *prev_size* ne smemo dostopati. Drugi najmanj pomemben bit se uporablja za označitev, ali je blok zaseden s sistemskim klicem `mmap`. Tretji najmanj pomemben bit pa označuje bloke, ki niso dodeljeni iz glavne arene. Zasedenost bloka se določi tako, da se s pomočjo polja *size* izračuna naslov polja *size* naslednika. In šele v naslednikovem polju *size* se preveri, ali je predhodnik zaseden.

```
struct malloc_chunk {
  size_t prev_size; // Mejna značka. Del prejšnjega
     bloka.
  size_t size; // Glava. Velikost bloka.
  /* Znotraj prostih blokov */
  struct malloc_chunk* fd; // Prejšnji prost blok
  struct malloc_chunk* bk; // Naslednji prost blok
  /* Znotraj večjih prostih blokov */
  struct malloc_chunk* fd_nextsize;
  struct malloc_chunk* bk_nextsize;
};
```



Slika 3.2: Zgradba bloka. Polje prev_size je mejna značka.

Polji *fd* in *bk* sta kazalca na prejšnji in naslednji prost blok v seznamu. Polji *fd_nextsize* in *bk_nextsize* pa sta kazalca na prejšnji in naslednji seznam prostih blokov. Na 64-bitni arhitekturi so vsa polja velikosti 8 B. Zato je za vsak zaseden blok 8 B presežka v obliki glave (polje *size*). A ker mora biti v prostem bloku dovolj prostora še za polja *prev_size*, *fd* in *bk*, velja, da mora biti vsebina najmanjšega bloka velikosti 24 B, kar skupaj z glavo znese 32 B. Na 32-bitni arhitekturi so vse te številke dvakrat manjše. Takšen način predstavitve je slab za majhne bloke, a je zato precej boljši za večje bloke, ker nima prevelike notranje fragmentacije.

Razporeditev blokov

Bloki so razvrščeni po velikosti na 128 seznamov. Prvih 64 seznamov vsebuje bloke od velikosti 8 B do 512 B z razmikom 8 B. Te bloke se tudi nekoliko drugače obravnava, ker imajo najmanjši možen razmik. Zato so znotraj vsakega seznama vsi bloki enake velikosti. Za večje bloke se razmik povečuje eksponentno. Za bloke, večje od 512 B, je 32 seznamov z razmikom 64 B. Temu sledi 16 seznamov z razmikom 512 B itd. Takšna organizacija je učinkovita, saj prevladujejo manjši bloki. Bloki se vstavljajo na seznam v vrstnem redu FIFO zaradi manjše fragmentacije. Pred vstavljanjem se za vsak blok še združi s prostima sosedoma. Vsi večji bloki in ostanki pri deljenju blokov so vstavljeni na poseben seznam.

Vrnjeni bloki, manjši od 160 B, so vstavljeni na enega izmed seznamov za

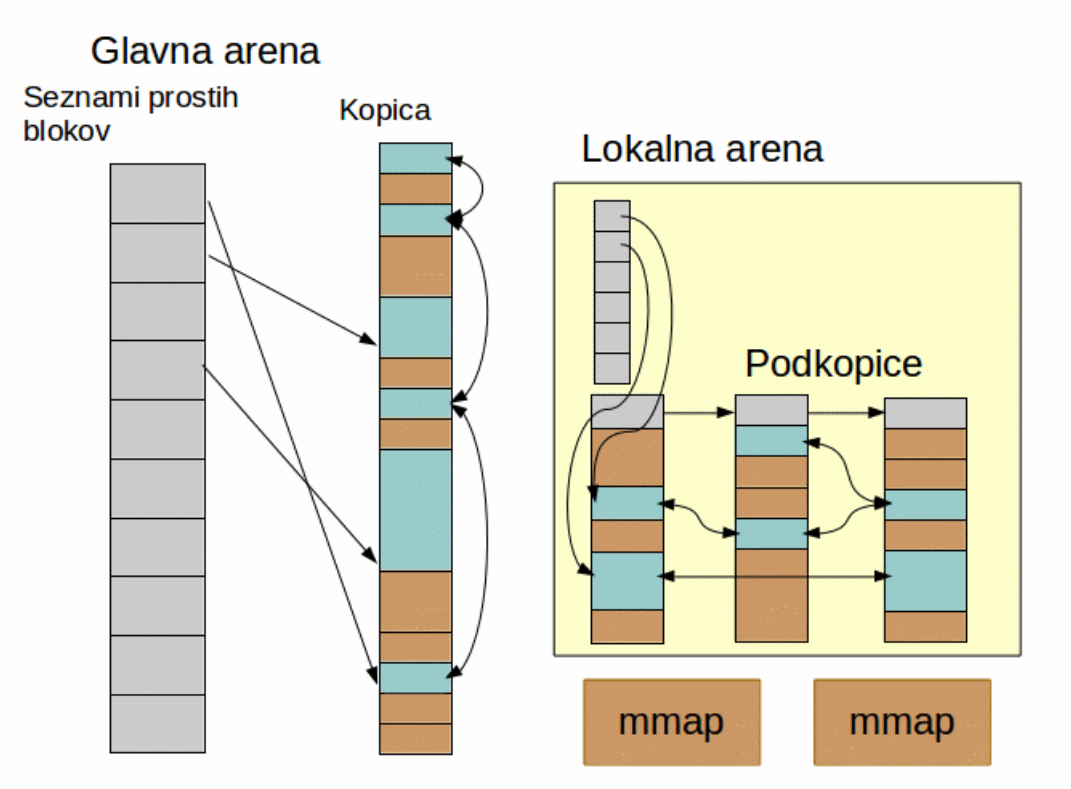
hitro vračanje. Bloki na tem seznamu ohranijo bit o zasedenosti in niso združeni s sosedoma. Bloki na seznamih za hitro vračanje se vnašajo v vrstnem redu LIFO zaradi večje verjetnosti, da bodo že v predpomnilniku. Ker bi se lahko zgodilo, da bloki s hitrih seznamov niso nikoli več uporabljeni ali združeni, se ti združijo v primeru, da pride zahteva za blok, večji od 64 KiB.

V vsaki areni se posebej obravnavata ostanek pri deljenju zadnjega bloka in vrh kopice. Ostanek pri deljenju (*last_remainder*) se uporablja zaradi boljše lokalnosti pri dodeljevanju, blok na vrhu kopice pa zato, ker ga je mogoče povečati. Blok na vrhu kopice (*top*) se uporabi zgolj v primeru, da ni najden prost blok. S tem se omogoči lažje vračanje pomnilnika sistemu. Blok na vrhu kopice mora biti vedno prost. Ta blok je v literaturi poimenovan divjina (angl. wilderness) [18].

3.2.2 Arene

Kopica je razdeljena na večje število aren. V vsakem primeru ima kopica glavno areno, ki se povečuje s pomočjo sistemskega klica `brk`. Preostale arene, imenovane lokalne arene, so ustvarjene v primeru, da ni nobene proste arene. Vsaka arena je predstavljena s strukturo *malloc_state*. V tej strukturi se nahajajo: polja do prostih blokov, ključavnica, ostanek pri zadnjem deljenju blokov (*top*) in kazalca na naslednjo areno ter na naslednjo prosto areno. Preostala polja za opis niso tako pomembna. Za vsako dodeljevanje in sproščanje pomnilnika je treba zakleniti ključavnico. Lokalne arene pridobijo pomnilnik z ustvarjanjem ali povečevanjem podkopic. Arene nimajo seznama podkopic, ampak ima vsaka podkopica kazalec na svojo areno. Nekoliko poenostavljeno je to predstavljeno s sliko 3.3.

Ali je blok iz globalne ali lokalne arene, se določi s pomočjo bita v glavi bloka. Če spada v glavno areno, je stvar rešena, sicer se poišče podkopico, kjer je kazalec na areno.



Slika 3.3: Zgradba GNUmalloca.

Podkopice

Podkopice nekoliko olajšajo pridobivanje in vračanje pomnilnika za lokalne arene. Velikost podkopic se lahko spreminja s pomočjo sistemskega klica `mmap`. Pogoj pri povečanju je ta, da ostane podkopica enoten kos pomnilnika (segment). Podkopice so najmanj 32 KiB in največ 32 MiB velike. Arene imajo po več podkopic, ker so le-te omejene velikosti. Podkopice so predstavljene v strukturi *heap_info*. Ključna polja v strukturi so kazalec na prejšnjo podkopico, kazalec na areno in velikost podkopice. Če podkopica ne vsebuje niti enega bloka, potem jo alokator vrne sistemu s sistemskim klicem `munmap`.

3.2.3 Dodeljevanje in sproščanje blokov

Pri dodeljevanju se najprej preveri zahtevana velikost. Če je ta nad 128 KiB, potem se blok dodeli neposredno s sistemskim klicem `mmap`. Sicer je postopek nekoliko bolj zapleten. Najprej nit poskuša zakleniti areno, ki jo je pred tem že uporabila za dodeljevanje. Če je arena zaklenjena ali v njej ni prostega pomnilnika in arene ni mogoče povečati, mora poiskati prosto areno ali ustvariti novo.

Iskanje prostega bloka je tu nekoliko bolj poenostavljeno opisano. Če je zahtevan blok dovolj majhen, preveri, ali je na seznamu blokov za hitro vračanje, sicer ga poišče na ustreznem seznamu prostih blokov. Iskanje ustreznega seznama prostih blokov poteka z uporabo bitnih operacij in je zelo hitro. Če oboje ne obrodi uspeha, preveri, ali je ostanek pri zadnjem deljenju bloka dovolj velik in ga vrne. Sicer mora poiskati med večjimi bloki in ga ustrezno razdeliti.

Vračanje blokov poteka takole. Če je blok dodeljen s sistemskim klicem `mmap`, se ga vrne neposredno sistemu z uporabo `munmap`. Preostali bloki se vrnejo na ustrezen seznam prostih blokov. Postopek vračanja je naslednji. Za blok se poišče areno. Če je blok dovolj majhen, se ga doda na seznam za hitro vračanje, sicer se preveri, ali ga je mogoče združiti s sosedoma. Združeni bloki niso vstavljeni na seznam z bloki, ki ustrezajo velikosti, ampak med preostale

bloke. Če je vrnjen blok večji od 64 KiB, potem se sproži združevanje blokov na seznamu za hitro vračanje. S tem se zmanjša fragmentacija in poveča možnost vračanja pomnilnika sistemu.

3.2.4 Zaključek

GNUmalloc je v osnovi dokaj preprost alokator, a je koda zaradi vseh optimizacij nekoliko manj berljiva. Alokator nima resnejših pomanjkljivosti in spada med najboljše alokatorje, ki poleg dodeljevanja omogoča tudi razhroščevanje, zbiranje statistik in nastavljanje parametrov. Dobro se obnese pri izvajanju enonitnih programov kot tudi večnitnih. Še največja pomanjkljivost je poraba pomnilnika v večnitnih programih, saj ima višjo porabo od ostalih alokatorjev, kar je problematično le za strežniške aplikacije.

3.3 tcmalloc

Razvoj tcmalloca sega v leto 2005, razvili so ga inženirji pri podjetju Google. Je tudi en izmed prvih alokatorjev, ki je namesto aren začel uporabljati predpomnilnik za nit[1]. Od opisanih alokatorjev se razlikuje tudi po tem, da je napisan v programskem jeziku C++.

3.3.1 Zgradba kopice

Kopica je pri tcmallocu razdeljena na osrednji predpomnilnik (angl. central cache), kopico strani (angl. page heap) in predpomnilnike za niti [14]. Kopica strani upravlja s prostimi stranmi, ki niso v predpomnilnikih. Stran ali večje število sosednjih si strani ima svojo strukturo *Span*. Takšni obsegi strani so znani tudi kot regije. Ne glede na to, da imajo praktično vsi sistemi stran, veliko 4 KiB, jo tcmalloc definira kot 8 KiB veliko. Posledica tega so nekoliko hitrejše operacije.

[1]Od tod tudi ime. tc je okrajšava za thread-cache

Kopica strani

V kopici strani se nahajajo proste regije. Proste regije so razdeljene na tiste, ki so bile vrnjene sistemu s sistemskim klicem `madvise`, in na tiste, ki niso bile vrnjene sistemu. Pomnilnik se sistemu vrača izključno s sistemskim klicem `madvise`.

Kopica strani je zadolžena za pridobivanje pomnilnika od sistema. Za to se uporabljata sistemska klica `brk` in `mmap`. Če en izmed sistemskih klicev ne uspe pridobiti pomnilnika, se uporabi drugega. Velikost blokov pri tem ne igra vloge. Najmanjša količina pomnilnika, ki jo alokator pridobi od sistema, je 1 MiB.

Za regije se uporablja strukturo *Span*. Regije so lahko različnih velikosti. V strukturi *Span* so naslednji podatki: kazalec na začetek regije, velikost regije v straneh, kazalec na seznam prostih blokov, velikost blokov, število zasedenih blokov in zastavica, ali je regija na seznamu prostih ali zasedenih. Poleg teh podatkov se hranijo še nekateri drugi.

Za vsak blok se da z uporabo funkcije *GetDescriptor* ugotoviti, kateri regiji pripada. Za preslikavo z naslova strani do ustrezne regije se uporablja podatkovno strukturo Radix drevo. Zato bloki ne uporabljajo glave, saj je velikost bloka mogoče določiti iz strukture *Span*.

Osrednji predpomnilnik

Če je v predpomnilniku za nit prevelika količina prostega prostora, so prosti bloki vstavljeni v osrednji predpomnilnik [14]. Tudi v primeru, da nit sprosti blok, ki je bil dodeljen neki drugi niti, je ta vrnjen v osrednji predpomnilnik. Pri vstavljanju in brisanju blokov iz osrednjega predpomnilnika se uporabljajo ključavnice. Prostor za nove bloke se pridobi iz kopic strani.

Bloki so razdeljeni na sezname. Kot majhne bloke se obravnava bloke, manjše od 1024 B. Ti so poravnani na 16 B in so razdeljeni v 64 razredov. Večji bloki imajo večji razmak. Dostop do pravega seznama se izračuna s pomočjo bitnih operacij. Bloki so tu razvrščeni na isti način kot v predpomnilniku za nit. V posameznem seznamu je lahko največ za 1 MiB prostih

blokov.

Predpomnilnik za nit

V vsakem predpomnilniku se zbira podatke o številu prostih blokov in o količini prostega prostora. Podatki za posamezno nit se hranijo v razredu *ThreadCache*. Predpomnilniki so povezani v seznam. V predpomnilniku je lahko največ 4 MiB prostega pomnilnika. Vsi skupaj pa imajo največ 32 MiB prostega pomnilnika. Predpomnilnik za nit dobi proste bloke iz osrednjega predpomnilnika.

3.3.2 Dodeljevanje in sproščanje

tcmalloc zelo malo uporablja ključavnice. Te se uporabljajo zgolj pri dostopu do osrednjega predpomnilnika in ob ustvarjanju novih niti. Pri dodeljevanju nit najprej poišče prosti blok v svojem predpomnilniku. Če ga ni na ustreznem seznamu, preveri, ali je v sosednjem seznamu večjih blokov morda na razpolago prost blok. Če ga ni, mora pridobiti proste bloke iz osrednjega predpomnilnika. Iz osrednjega predpomnilnika je vrnjenih večje število blokov in ne samo en.

Če nit sprošča svoje bloke, potem so ti vrnjeni v njen predpomnilnik, sicer morajo biti vrnjeni v osrednji predpomnilnik. tcmalloc blokov ne združuje, ker so znotraj regije vsi enake velikosti.

3.3.3 Zaključek

tcmalloc spada med najhitrejše alokatorje in ima zelo nizko porabo pomnilnika. Poleg dodeljevanja ima zelo dobro podporo razhroščevanju in profiliranju. Glede tega je morda celo najboljši. Le glede nastavljivosti nima tako dobrih opcij kot nekateri drugi alokatorji. Zaradi vseh teh lastnosti je tcmalloc eden izmed priljubljenejših alokatorjev.

3.4 Lockless

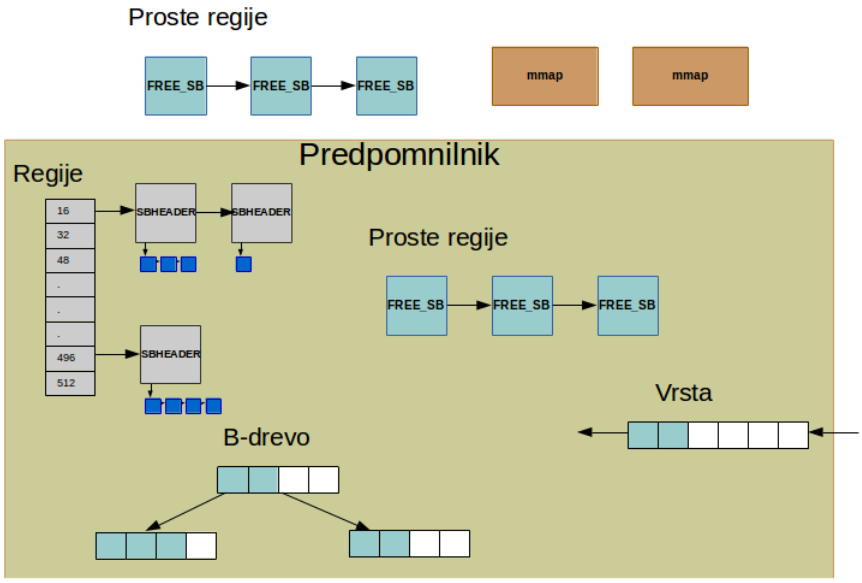
Lockless ali llalloc je razmeroma neznan alokator, ki ga je napisal Steven Von Fuerst leta 2009. Lockless za dodeljevanje in sproščanje uporablja predpomnilnik za nit, zato ne rabi ključavnic za delovanje [4].

3.4.1 Zgradba kopice

Bloki do velikosti 512 B se nahajajo v regijah. Znotraj vsake regije so vsi bloki iste velikosti. Prosti bloki znotraj regije so povezani v seznam. Vse regije so velike 64 KiB ne glede na velikost blokov v njih. Prostor za regije se pridobi s sistemskim klicem `brk`. Ko v regiji ni več zasedenega bloka, je vstavljena med proste regije. Za regijo se uporablja struktura *sbheader*. Do regije lahko dostopa le ena nit, zato uporaba ključavnic ni nujna. Regije so razvrščene na 32 seznamov, na vsakem seznamu pa so regije z bloki iste velikosti. Z vsakim seznamov se velikost blokov poveča za 16 B. Torej za vse manjše bloke velja, da so poravnani na 16 B. Regije so razdeljene na proste, delno zasedene in polne.

Podatki za vsako nit posebej se hranijo v strukturi *atls*. Tej strukturi lahko rečemo tudi predpomnilnik. Ko ima predpomnilnik 64 ali več prostih regij, jih označi kot proste s sistemskim klicem `madvise`. Nato so te regije vstavljene na seznam, dostopen vsem nitim. Pri tem je treba uporabiti ključavnico, da se zaklene dostop do seznama. A ker se to ne dogaja pogosto, nima velikega vpliva na izvajanje. Da slučajno ne bi vstavljanje ali brisanje s seznama povzročilo predolgega čakanja, je globalno dostopnih seznamov enako številu jeder procesorja. Na sliki 3.4 je predstavljena zgradba alokatorja.

Bloki, večji od 512 B in manjši od 256 MiB, se nahajajo v B-drevesu. Zasedeni bloki te velikosti imajo v glavi zapisano velikosti. Prosti bloki pa izkoriščajo prazen prostor za strukturo *btree*. Prostor za te bloke je pridobljen s sistemskim klicem `mmap`. Ker so operacije iskanja, vstavljanja in brisanja nad B-drevesi časovne zahtevnosti $O(\log n)$, uporablja B-drevo predpomnil-



Slika 3.4: Zgradba kopice.

nik za pogoste zahteve. Zato je časovna zahtevnost omenjenih operacij lahko tudi konstantna [4]. Če so najdeni bloki v drevesu preveliki, so razdeljeni, preostanek pa je vnesen nazaj v drevo.

Bloki, večji od 256 MiB, se pridobijo direktno s klicem `mmap` in so direktno vrnjeni sistemu.

Sočasne vrste

Ker lahko neka nit sprosti blok, dodeljen neki drugi niti, in se morajo bloki vrniti v svojo regijo, je treba zagotoviti premikanje blokov med predpomnilniki. Zato ima vsaka nit sočasno vrsto, ki ne rabi ključavnice za vstavljanje ali brisanje [4]. Vse niti imajo pravico vstavljanja blokov v vrsto, le nit, ki je lastnica vrste, pa jih lahko pobriše. V vrsto so vstavljeni vsi bloki z izjemo tistih, ki so večji od 256 MiB.

Ko nit preneha delovati, se njen predpomnilnik (struktura *atls*) združi s predpomnilnikom aktivne niti. S tem se izognemo primeru, da bi nit, ki je prenehala delovati, prejemala v svojo vrsto proste bloke. Teh blokov ne bi mogel nihče več uporabiti, zato je združevanje vrst pomembno.

3.4.2 Dodeljevanje in sproščanje blokov

Ker je dodeljevanje blokov nad 512 B relativno preprosto, je tu podan opis dodeljevanja in sproščanja zgolj za majhne bloke. Ob prvem dodeljevanju se za nit ustvari struktura *atls*. Pri dodeljevanju se poišče regijo z bloki zahtevane velikosti. Dodeljevanje iz regije se zgodi v konstantnem času, saj zahteva le povečanje kazalca ali brisanje prvega bloka s seznama. Če ni take regije, se iz vrste, kamor proste bloke vstavljajo druge niti, poberejo prosti bloki. Če med prostimi bloki ni nobenega zahtevane velikosti, potem mora nit na neki način dobiti novo regijo. Najprej jo poišče v svojem predpomnilniku, če je ne najde, se preveri v globalno dostopnem seznamu prostih regij. Če proste regije nikjer ni, potem od sistema zahteva dodaten pomnilnik in se ustvari nova regija.

Pri sproščanju blokov je za blok treba ugotoviti njegovo velikost in pripadajoč predpomnilnik. Ali je blok majhen, se ugotovi tako, da se preveri, ali je naslov znotraj segmenta, pridobljenega s sistemskim klicem `brk`. Če ni, potem ima blok v glavi zapisano velikost. Če je nad 256 MiB, je vrnjen sistemu. Za majhen blok se iz strukture *sbheader* določi, ali nit sprošča svoj blok ali blok, dodeljen neki drugi niti. Če blok pripada drugi niti, je vstavljen v vrsto prostih blokov te niti. Tudi bloki, ki so večji od 512 B, so vstavljeni v to vrsto. Sicer ista nit, kot je pridobila blok, tudi vrača, zato ga doda med proste bloke v pripadajočo regijo. Če je to bil zadnji zaseden blok v regiji, se regijo doda na seznam prostih regij.

3.4.3 Zaključek

Lockless se je na testih presenetljivo dobro obnesel in je en izmed najboljših alokatorjev. Na hitrostnih testih se je obnesel odlično, medtem ko pri merjenju porabe nekoliko slabše, a še vedno zelo dobro. Razlog za hitrost najbrž tiči v tem, da imajo podatki zaradi predpomnilnika za nit dobro lokalnost. Poleg tega se skoraj v celoti izogne zaklepanju. Poleg dodeljevanja omogoča še razhroščevanje. Na žalost zbiranje statistik ni delovalo in je prekinilo

delovanje programa. Poleg tega ni podprto nastavljanje parametrov.

3.5 Preostale implementacije

Poleg zgoraj opisanih je še kar nekaj alokatorjev, ki se uporabljajo, a tu niso opisani. jemalloc je poleg tcmalloca najbrž najpogosteje predlagan kot zamenjava za GNUmalloc. Je en izmed najhitrejših in najučinkovitejših, kar zadeva porabo. Poleg tega ima dobro podporo razhroščevanju in nastavljivosti. Med najboljše alokatorje spada tudi Hoard, ki je bil en izmed prvih alokatorjev z arenami, a je nato prešel kombinacijo aren in predpomnilnika za nit. Hoard je tudi en izmed najbolj modularnih alokatorjev in programerju omogoča relativno enostavno prilaganje kode alokatorja. Alokator, ki postavlja varnost na prvo mesto, je ottomalloc [20]. Po hitrosti in prenosljivosti se ne more primerjati s preostalimi, a nudi nekoliko več varnosti v primeru, da ima program še neodkrite varnostne pomanjkljivosti.

3.6 Razvrstitev alokatorjev

V tabeli 3.2 so alokatorji na preprost način razvrščeni. Poleg opisanih alokatorjev so vključeni tudi alokatorji z meritev.

Ime	**Zgradba kopice**	**Shranjevanje blokov**	**Uporaba `mmap`**
dlmalloc	Enotna kopica	Seznam	Za velike bloke
phkmalloc	Enotna kopica	Polje bitov in seznam	Zgolj za interno rabo
GNUmalloc	Arene	Seznam	Za velike bloke in za arene
jemalloc	Arene in predpomnilniki	Polje bitov in seznam	Za vse
tcmalloc	Predpomnilniki	Seznam	Če `brk` odpove
Hoard	Arene in predpomnilniki	Seznam	Za vse
Lockless	Predpomnilniki	Seznam	Za vse razen majhnih blokov
ottomalloc	Enotna kopica	Polje bitov in seznam	Za vse

Tabela 3.2: Razvrstitev implementacij glede na delovanje.

Poglavje 4

Primerjava hitrosti in učinkovitosti

Najverjetneje sta glavna razloga za zamenjavo obstoječega alokatorja hitrost in manjša poraba pomnilnika. Medtem ko je merjenje hitrosti dokaj enostavno, je merjenje porabe nekoliko bolj zapleteno. Primerjave se lahko izvede na testnih programih ali pa na navadnih programih, ki dovolj pogosto kličejo funkcije za dodeljevanje pomnilnika. Prednost testnih programov je ta, da je lažje razumeti njihovo izvajanje in da jim nastavljamo določene parametre. Zato so uporabni za odkrivanje prednosti in slabosti alokatorjev. Po drugi strani pa navadni programi dajo realno sliko, a za te programe alokator nima dovolj velikega vpliva ali pa je čas izvajanja takih programov lahko odvisen od zunanjih faktorjev in jih zato ne moremo vključiti v primerjavo.

Na sistemih Unix je enostavno zagnati program z drugačnim alokatorjem. V ukazni vrstici je treba pred izvajanjem v okoljsko spremenljivko z imenom *LD_PRELOAD* vnesti pot do knjižnice novega alokatorja.

```
LD_PRELOAD="/usr/local/lib/libjemalloc.so" ./program
```

Na meritvah ni prisoten phkmalloc, ker je ob uporabi povzročil sesutje programa. Na meritvah hitrosti delovanja ni prisoten ottomalloc, ker je bil nekajkrat počasnejši od preostalih alokatorjev. Ni povsem jasno, ali je to posledica tega, da je ottomalloc pisan za sistem OpenBSD in zato ni prenosljiv.

Meritve so bile opravljene na računalniku s procesorjem Intel Core i5-4570 3,20GHz, s 16 GiB pomnilnika in z operacijskim sistemom Linux z jedrom verzije 3.13.

4.1 Testni program

Testni program uporablja preproste podatkovne strukture, kot sta povezan seznam in dvojiško iskalno drevo. Lokalnost podatkov je bolj pomembna za seznam kot za drevo, zato se v testnem programu več uporablja seznam. Na začetku glavna nit ustvari večje število blokov in jih vstavi na seznam, nato ustvari niti, ki pobrišejo vsaka enak delež blokov s seznama. Nato vsaka nit opravi pet ponovitev naslednjih dejanj. Ustvari določeno število blokov in jih vstavi na seznam. Te podatke nato desetkrat uporabi in cel seznam sprosti. Nato zasede določeno število tabel in jih prepiše z ničlami. Za konec še vstavi določeno število elementov v iskalno drevo.

Izkazalo se je, da pisanje v podatkovne strukture nekatere alokatorje bolj upočasni kot druge. Zato je pomembno, da se vsak zaseden blok uporabi in ne le obdrži za nekaj časa in nato sprosti. Količina dela za testni program je fiksna, zato se z večanjem števila niti delo porazdeli. Najbrž pa je v testnem programu uporabljenih premalo blokov različnih velikosti, kar se pokaže pri merjenju, ko se upošteva lokalnosti podatkov.

4.2 Merjenje časa

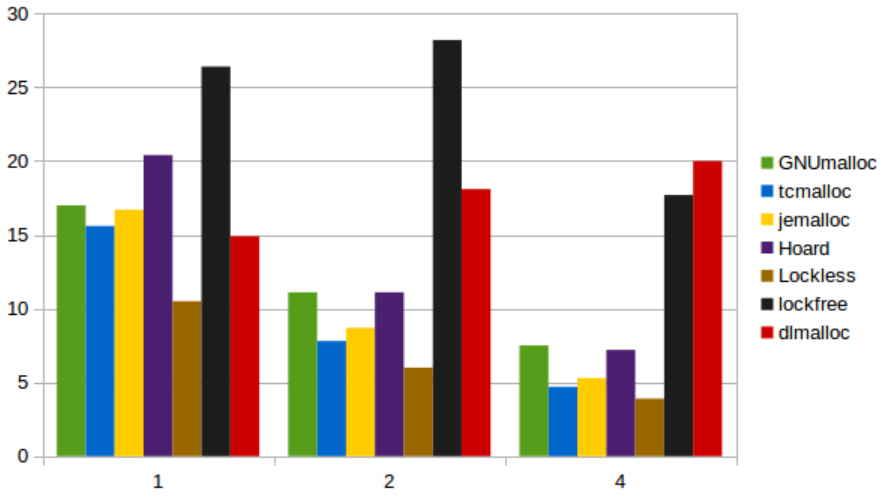
Za vsak program lahko izmerimo skupen čas izvajanja in čas, porabljen za dodeljevanje in sproščanje pomnilnika. Primerjava alokatorjev po skupnem času izvajanja je boljša od primerjave časa, porabljenega za dodeljevanje in sproščanje. Razlog za to je ta, da nekateri alokatorji dosegajo boljšo lokalnost blokov, zato je izvajanje programske kode zunaj funkcij za dodeljevanje pomnilnika hitrejše.

4.2.1 Pohitritev z večanjem števila niti

Z večanjem števila niti se pričakuje od programov, da bodo hitreje delovali. Pohitritev je možna le, če je število niti manjše od števila jeder procesorja oziroma od števila logičnih jeder. Žal računalnik, kjer so bile izvedene meritve, ne more poganjati več kot štirih niti sočasno. Da bi dobili boljši vpogled, bi morali pognati vsaj 8 ali celo 16 niti sočasno. Teoretična pohitritev je nekoliko nižja od števila štiri, ker na začetku samo ena nit pridobiva pomnilnik.

Implementacija	**1**	**2**	**4**	**Pohitritev**
GNUmalloc	17,0	11,1	7,5	2,3
tcmalloc	15,6	7,8	4,7	3,3
jemalloc	16,7	8,7	5,3	3,7
Hoard	20,4	11,1	7,2	2,8
Lockless	10,5	6,0	3,9	2,7
lockfree	26,4	28,2	17,7	1,5
dlmalloc	14,9	18,1	20,0	0,7

Tabela 4.1: Časi izvajanja za različno število niti.



Slika 4.1: Čas izvajanja prikazan v grafu.

V tabeli 4.1 so v stolpcih 1, 2 in 4 časi izvajanja v sekundah. Iz tabele 4.1 in grafa 4.1 je razvidno, da je največjo pohitritev dosegel jemalloc, kar pa ne pomeni, da bi se z večanjem števila niti odrezal najbolje. Najboljši čas je dosegel Lockless, ki ne uporablja ključavnic. Na primeru dlmalloca vidimo, da na večnitnih programih, ki veliko uporabljajo funkcijo malloc, ni mogoče doseči pohitritve, če je uporabljen alokator z enotno kopico.

4.2.2 Lokalnost dodeljenih blokov

Lokalnost dodeljenih blokov morda ni tako pomembna kot pohitritev z večanjem števila niti, a je vseeno nekolika prezrta v literaturi. Testni program uporablja dve podatkovni strukturi: seznam in rdeče-črno drevo. Ker za drevesa ni realno pričakovati dobre lokalnosti, se je le-ta merila zgolj na seznamu. Morda je zaradi tega meritev nekoliko pomanjkljiva, a so se kljub temu pokazale razlike med alokatorji.

Implementacija	**1**	**10**	**100**	**Upočasnitev**
GNUmalloc	4,9	7,5	36,3	7,4
tcmalloc	3,4	4,7	19,1	5,6
jemalloc	4,3	5,3	19,3	4,5
Hoard	6,4	7,2	20,1	3,1
Lockless	2,5	3,9	18,3	7,3
lockfree	2,5	17,7	102,8	41,1
dlmalloc	23,0	20,0	71,5	3,1

Tabela 4.2: Časi izvajanja za različno število obdelav blokov.

V tabeli 4.2 so v stolpcih 1, 10 in 100 časi izvajanja v sekundah za dano število obdelav seznama. V zadnjem stolpcu je podana upočasnitev od ene obdelave do stotih obdelav za posamezen alokator. Obdelava pomeni sprehod skozi seznam, pri tem pa se podatki preberejo, spremenijo in nazaj zapišejo v strukturo. Časi za alokatorje tcmalloc, jemalloc, Hoard in Lockless

z večanjem števila obdelav seznama konvergirajo in imajo podatki praktično isto lokalnost. Glede tega se nekoliko slabše obnese GNUmalloc, ki pri tem poveča razliko. Zelo slabo pa se obnese lockfree, ki ne zagotavlja praktično nobene lokalnosti dodeljevanj. Za dlmalloc kljub majhni upočasnitvi bi težko rekli, da zagotavlja dobro lokalnost, saj za delovanje porabi preveč časa [1].

4.2.3 Čas izvajanja na različnih programih

V tabeli 4.3 so časi izvajanja nekaterih programov. Za program sed je bil izmerjen le čas izvajanja, saj je bila poraba prenizka, za perl in gcc je v levem stolpcu čas izvajanja, v desnem poraba. Skripta v programskem jeziku perl je izkoristila vsa štiri jedra. Programa gcc in sed pa sta enonitna. Pri izvajanju skripte iz jezika perl izstopa le dlmalloc, ki je najpočasnejši in nekoliko presenetljivo z najnižjo porabo. Pri prevajanju kode z gcc sta se izmed tistih, ki uporabljajo arene ali predpomnilnik, le GNUmalloc in Lockless dobro obnesla. Lockless je celo prehitel dlmalloc. Program malo uporablja dodeljevanje pomnilnika, zato so razlike majhne, kjer le jemalloc malenkost odstopa.

4.3 Merjenje fragmentacije

Fragmentacijo pomnilnika lahko izmerimo na več različnih načinov. V članku [15] so opisane štiri mere fragmentacije. Skupna poraba pomeni količino zasedenega pomnilnika, kar vključuje vse podatkovne strukture alokatorja in prostor, ki ni vrnjen OS. Dejanska poraba pa pomeni predstavlja količino pomnilnika, ki ga je program pridobil od alokatorja. Skupna poraba je vedno večja ali enaka dejanski porabi.

1. Skupna poraba relativna na dejansko porabo programa skozi ves čas izvajanja. Slabost tega postopka je, da se špice, ko je poraba največja,

[1] Iz tabele je razvidno, da je dlmalloc porabil manj časa za deset obdelav kot za eno. To ni napaka. Podobno se je zgodilo za ottomalloc, kar je najbrž posledica tega, da imata oba enotno kopico. Rezultatov za ottomalloc ni v tabeli, ker na sistemih Linux najverjetneje ne deluje tako, kot so si avtorji zamislili.

Implementacija	**perl**		**gcc**		**sed**
GNUmalloc	22,0	419	59,9	94	4,0
tcmalloc	20,1	377	66,7	101	3,9
jemalloc	23,1	413	63,1	95	4,3
Hoard	22,1	471	63,7	105	3,9
Lockless	20,9	429	58,0	100	4,0
dlmalloc	33,0	360	59,0	95	4,0

Tabela 4.3: Čas izvajanja za različne programe. V levem stolpcu je čas v sekundah in v desnem poraba pomnilnika v MiB. Za sed je bil izmerjen zgolj čas izvajanja.

omilijo zaradi večjega časovnega obdobja nižje fragmentacije. Poleg tega jo je težje izmeriti.

2. Skupna poraba relativna na največjo dejansko porabo v trenutku dejanske največje porabe pomnilnika. To ni nujno trenutek, ko je skupna poraba pomnilnika največja, zato ta mera ni najbolj informativna.

3. Skupna poraba relativna na dejansko porabo v trenutku največje skupne porabe pomnilnika. Ta mera je smiselna, saj pokaže fragmentacijo v času največje porabe pomnilnika.

4. Največja skupna poraba relativna na največjo dejansko porabo. Ni nujno, da sta to istočasna dogodka.

4.4 Merjenje porabe pomnilnika

Porabo pomnilnika lahko merimo na več načinov. En pristop je ta, da jo merimo znotraj programa ob vsakem klicu funkcij za dodeljevanje pomnilnika. Drugi pristop je tak, da merimo porabo na določen časovni interval z nekim drugim programom. Prvi pristop je uporaben le za programe, kjer lahko spremenimo kodo in jih nato prevedemo. Ima pa to prednost, da je na

abscisni osi namesto časa število klicev funkcij za dodeljevanje pomnilnika ali pa količina dodeljenega pomnilnika. S tem so primerjave med različnimi implementacijami lažje, ker se čas izvajanja lahko razlikuje. Po drugi strani pa je merjenje zunaj programa brez vpliva na izvajanje le-tega in pokaže porabo v času.

Operacijski sistem meri porabo za vsak program v navideznem naslovnem prostoru in v fizičnem pomnilniku. Poraba v navideznem naslovnem prostoru je lahko zelo visoka in ne odraža dejanske porabe, prav tako pa nima vpliva na porabo v fizičnem pomnilniku. Zato je primerneje meriti porabo fizičnega pomnilnika, ki pa ima to slabost, da ne meri porabe izmenjevalnega prostora na disku. V obeh primerih pa se ne meri samo velikosti kopice, ampak tudi tekst programa, sklad in deljene knjižnice.

Podatke o posameznem procesu lahko pridobimo iz datotek v direktoriju `/proc/[pid]/`. Podatki o porabi se nahajajo v datoteki `statm`, kjer je šest različnih številk v isti vrstici, za enoto pa se uporablja velikost strani. Prva predstavlja porabo v navideznem naslovnem prostoru in se imenuje `VmSize`, druga pa predstavlja porabo fizičnega pomnilnika in se imenuje `VmRSS` [10]. Te številke bi se sicer dalo pridobiti tudi s funkcijo `getrusage`, a žal ne na sistemu Linux.

4.4.1 Meritev

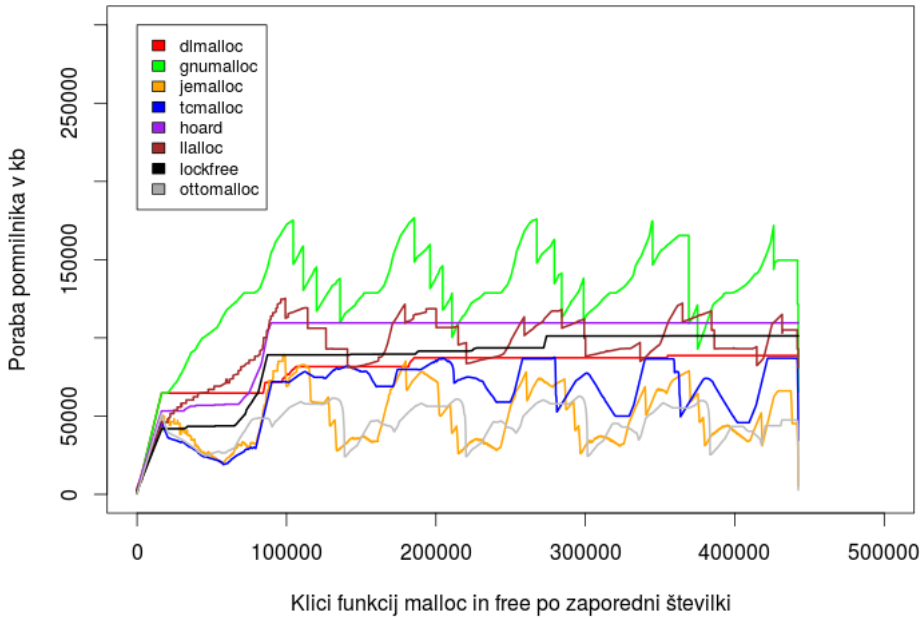
Meritev je bila opravljena na testnem programu z nekoliko manjšim številom dodeljenih blokov kot na testu hitrosti izvajanja. V tabeli 4.4 so podani podatki za povprečno in največjo porabo programa v MiB. Sicer je za programe, ki imajo kratek čas izvajanja, pomembnejša največja poraba. A ker nas zanima, kako učinkovito alokatorji vračajo pomnilnik sistemu, ima zaradi tega nekoliko večjo težo povprečna poraba.

Na grafu 4.2 je razvidna poraba programa skozi čas izvajanja. Na abscisni osi so zaporedni klici funkcij za dodeljevanje ali sproščanje pomnilnika. Na ordinatni osi pa je poraba v MiB.

Pri merjenju porabe se je najbolje obnesel ottomalloc, ki pa ni optimizi-

Implementacija	Povprečna poraba	Največja poraba
GNUmalloc	129,7	176,7
tcmalloc	64,0	87,6
jemalloc	49,5	89,1
Hoard	98,7	109,5
Lockless	93,5	125,3
lockfree	85,5	101,3
ottomalloc	45,1	62,7
dlmalloc	80,4	88,7

Tabela 4.4: Poraba pomnilnika v MiB za različno implementacijo.



Slika 4.2: Graf porabe pomnilnika.

ran za hitrost in porabi veliko časa za vračanje. Le nekoliko slabše se obnese jemalloc, ki pa je en izmed najhitrejših alokatorjev. Glavna prednost jemal-

loca je ta, da ima zelo nizko fragmentacijo in da učinkovito vrača pomnilnik sistemu. Poleg omenjenih dveh ima nizko porabo tudi tcmalloc, ki ravno tako učinkovito vrača pomnilnik sistemu. dlmalloc ima nizko fragmentacijo ob najvišji porabi, a ne "zna" vračati pomnilnika sistemu, kar je posledica enotne kopice.

Najslabše se je obnesel GNUmalloc. Razlog morda tiči v tem, da alokator ni izkoristil prostih blokov iz glavne arene, kar je razvidno iz začetka grafa 4.2, kjer so preostali alokatorji po začetnem vzponu porabe imeli padec. Le Lockless in GNUmalloc sta porabila dodaten prostor. Ta anomalija je posledica tega, da je glavna nit pridobivala bloke, preostale pa so jih sproščale. Takšni programi so redkejši. A kljub temu je to resna pomanjkljivost GNUmalloca. Tudi preostali alokatorji imajo svoje pomanjkljivosti glede porabe, ki na tej meritvi niso prišle tako do izraza kot pomanjkljivosti GNUmalloca.

4.5 Vpliv napačne skupne rabe na hitrost

Implementacija	**Cache-scratch**	**Cache-thrash**
GNUmalloc	2,7	2,7
tcmalloc	50,6	50,0
jemalloc	2,7	2,8
Hoard	38,8	2,7
Lockless	2,7	2,7
lockfree	2,7	2,8
dlmalloc	17,2	5,4

Tabela 4.5: Čas izvajanja s programoma Cache-scratch in Cache-thrash, ki povzročata napačno skupno rabo.

Napačna skupna raba je sicer v programih dokaj redka, a ko do nje pride, se izvajanje programa bistveno upočasni. Iz tabele 4.5 lahko vidimo, da napačno skupno rabo povzročita tcmalloc in Hoard, medtem ko jo dlmalloc

povzroči v manjšem obsegu. V preostalih implementacijah do napačne skupne rabe ne pride. Napačna skupna raba se meri na programih Cache-scratch in Cache-thrash.

4.6 Meritve zgrešitev predpomnilnika in TLB

S podatki iz meritev zgrešitev predpomnilnika in TLB lahko ugotovimo, ali so zgrešitve predpomnilnika ozko grlo pri delovanju programa. Ker se delež zgrešitev računa glede na število nalaganj podatkov iz predpomnilnika, so v stolpcu L1-loads zapisana števila nalaganj v milijardah. Zgrešitve podatkovnega TLB so zapisane v stolpcu dTLB-misses in predstavljajo delež zgrešitev glede na nalaganja predpomnilnika. Meritve so bile opravljene s programom perf. Žal se podatkov za zgrešitve tretjega nivoja predpomnilnika ni dalo pridobiti.

Implementacija	**L1-loads**	**L1-misses**	**L1-stores**	**dTLB-misses**
GNUmalloc	18,5	6,2%	12,4	0,5%
tcmalloc	15,8	5,8%	8,5	1,0%
jemalloc	26,3	3,2%	12,4	0,3%
Hoard	19,7	4,3%	10,2	0,5%
Lockless	11,7	6,8%	6,6	0,7%
lockfree	12,2	19,9%	7,0	6,0%
dlmalloc	48,1	5,2%	26,0	0,6%

Tabela 4.6: Zgrešitve predpomnilnika in TLB.

Iz tabele 4.6 je razvidno, da z izjemo enega alokatorja noben ne izstopa pri zgrešitvah predpomnilnika. Nekoliko presenetljivo ima jemalloc visoko število nalaganj v predpomnilnik in se kljub temu dobro izkaže na testu iz lokalnosti podatkov, ker sta GNUmalloc in dlmalloc relativno podobna. Lahko potrdimo, da arene zmanjšajo število nalaganj predpomnilnika in zmanjšajo možnost lažne skupne rabe.

4.7 Število sistemskih klicev

Približen oris, kako alokatorji delujejo, dobimo tudi s tem, ko zmerimo št. uporabljenih sistemskih klicev med delovanjem. V tabeli 4.7 so podatki za program, ki je bil zagnan z istimi parametri kot v primeru merjenja časa. Čas izvajanja programa se deli na uporabniški in sistemski čas. Pod sistemski čas spada izvajanje sistemskih klicev kot tudi zaklepanje in odklepanje ključavnic. Vsi alokatorji imajo delež sistemskega časa manjši od 10% z izjemo dlmalloca, ki ima delež sistemskega časa približno 20%. Iz tega bi lahko sklepali, da nekoliko večje število sistemskih klicev ne bistveno upočasni delovanja alokatorja. Na sistemih Linux se sistemske klice meri s programom `strace`.

Implementacija	**brk**	**mmap**	**munmap**	**madvise**
GNUmalloc	2427	60	65	48
tcmalloc	313	29	1	118959
jemalloc	1	94	7	101097
Hoard	1	5673	13	4
Lockless	69	34	1	17644
lockfree	1	2368	177	4
dlmalloc	132718	19	1	4

Tabela 4.7: Število uporabljenih sistemskih klicev za vsako implementacijo.

Poglavje 5

Sklep

V delu so bili opisani različni načini dodeljevanja pomnilnika in alokatorji sami. Meritve so pokazale, da so odstopanja po času izvajanja manjša od razlik v zgradbi ali načinu delovanja alokatorjev. K temu največ pripomore to, da so alokatorji produkt več let optimizacij in popravkov. Na meritvah so poleg GNUmalloca dosegli dobre rezultate še tcmalloc, jemalloc, Lockless in Hoard. Čeprav so tcmalloc, jemalloc in Lockless v večnitnih programih prehiteli GNUmalloc, so v enonitnih programih še vedno nekoliko počasnejši. Na testih je bil uporabljen računalnik s štirijedrnim procesorjem. Nedvomno bi se na procesorju z večjim številom jeder povečale razlike med alokatorji.

V prihodnosti bo pomembno, kako bo večanje števila jeder vplivalo na hitrost izvajanja, saj se število jeder z leti povečuje. Morda so danes razlike med alokatorji še dokaj majhne, ker ima večina sistemov 4 ali 8 jeder. Ali bodo razmerja ostala enaka, tudi ko bodo uporabljeni procesorji s 64 ali 128 jedri, pa bo pokazal čas.

Za večino uporabnikov je izbira alokatorja nepomembna. Za programerje je morda celo pomembnejše, kaj alokator nudi za razhroščevanje programov ali za iskanje ozkega grla pri izvajanju. Področje, na katerem je izbira alokatorja pomembna, so strežniške aplikacije kot npr. podatkovne baze ali spletni strežniki. Te so najbolj požrešne pri porabi pomnilnika, zato so tu možne večje razlike v delovanju.

Literatura

[1] Address space layout randomization. Dosegljivo: `https://en.wikipedia.org/wiki/Address_space_layout_randomization`. [Dostopano 20.8.2016].

[2] Buddy memory allocation. Dosegljivo: `https://en.wikipedia.org/wiki/Buddy_memory_allocation`. [Dostopano 20.8.2016].

[3] Intel haswell. Dosegljivo: `http://www.7-cpu.com/cpu/Haswell.html`. [Dostopano 20.8.2016].

[4] Lockless. Dosegljivo: `http://locklessinc.com/technical_allocator.shtml`. [Dostopano 20.8.2016].

[5] madvise(2). Dosegljivo: `http://linux.die.net/man/2/madvise`. [Dostopano 20.8.2016].

[6] Memory hierarchy. Dosegljivo: `https://en.wikipedia.org/wiki/Memory_hierarchy`. [Dostopano 20.8.2016].

[7] Memory segmentation. Dosegljivo: `https://en.wikipedia.org/wiki/Memory_segmentation`. [Dostopano 20.8.2016].

[8] mmap(2). Dosegljivo:
`http://linux.die.net/man/2/mmap`. [Dostopano 20.8.2016].

[9] Region-based memory management. Dosegljivo:
`https://en.wikipedia.org/wiki/Region-based_memory_management`. [Dostopano 20.8.2016].

[10] Resident set size. Dosegljivo:
`https://en.wikipedia.org/wiki/Resident_set_size`. [Dostopano 20.8.2016].

[11] sbrk(2). Dosegljivo:
`http://linux.die.net/man/2/sbrk`. [Dostopano 20.8.2016].

[12] Programming languages - c. Dosegljivo:
`http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf`, 2011. [Dostopano 20.8.2016].

[13] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. *SIGPLAN Not.*, 35(11):117–128, November 2000.

[14] James Golick. How tcmalloc works. Dosegljivo:
`http://jamesgolick.com/2013/5/19/how-tcmalloc-works.html`. [Dostopano 20.8.2016].

[15] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: Solved? *SIGPLAN Not.*, 34(3):26–36, October 1998.

[16] Poul-Henning Kamp. Malloc(3) revisited. Dosegljivo:
`http://phk.freebsd.dk/pubs/malloc.pdf`. [Dostopano 20.8.2016].

[17] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[18] Doug Lea. A memory allocator. Dosegljivo: `http://g.oswego.edu/dl/html/malloc.html`, 1996. [Dostopano 20.8.2016].

[19] Maged M. Michael. Scalable lock-free dynamic memory allocation. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 35–46, New York, NY, USA, 2004. ACM.

[20] Otto Moerbeek. A new malloc(3) for openbsd. Dosegljivo: `https://openbsd.id/papers/eurobsdcon2009/otto-malloc.pdf`. [Dostopano 20.8.2016].

[21] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[22] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of the International Workshop on Memory Management*, IWMM '95, pages 1–116, London, UK, UK, 1995. Springer-Verlag.